Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Detección de personas en secuencias de imágenes en entornos interiores

Autor: Antonio Del Abril De Mur

Tutor: Cristina Losada Gutiérrez y Marta Marrón Romera

2017

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Detección de personas en secuencias de imágenes en entornos interiores

Autor: Antonio Del Abril De Mur

Director: Cristina Losada Gutiérrez y Marta Marrón Romera

Tribunal:

Presidente: Javier Macías Guarasa

Vocal 1º: Elena López Guillén

Vocal 2º: Cristina Losada Gutiérrez

Calificación: .....

Fecha: .....

A mis padres, Antonio y Concha, por darme todo lo que tienen para que cada día me convierta en mejor persona, por ser el apoyo constante e incodicional que tengo y tendré siempre. A mis hermanos, Alberto y Laura, por ser mi referente, mi ejemplo a seguir, por ser además amigos y protectores. Porque sin ellos no podría ser quien soy ahora. A mis compañeros y amigos: Ismael, Pablo, Adrián, Clara, Álvaro, Jorge, Carlos, Fernando, Javier, Carlos, Óscar, Gonzalo, David, Cristina, Lidia, Pedro y unos largos puntos suspensivos. Por poder contar con ellos y haber pasado buenos y malos momentos juntos, por ser la calma y el desahogo que necesitaba en muchos momentos, y el apoyo y consuelo en otros. Al resto de familiares y amigos que no recordé en el momento de escribir esto, pero que también se merecen una mención especial, ustedes saben quienes son y todo lo que han hecho por mí. A mis profesores, por hacerme crecer personal y profesionalmente.

A todos vosotros, gracias, sin cada una de las pequeñas y grandes cosas que me habéis aportado no sería lo que soy, y es algo de lo que me siento orgulloso.

Resumen

El objetivo de este proyecto es la mejora del algoritmo de detección y seguimiento de personas en imágenes 2D de color en escenas de videovigilancia, ya implementado por Marcos Baptista Ríos en su Trabajo Fin de Grado [3].

Esta mejora se basa en incluir la detección de la mitad superior de las personas, necesaria para la identificación de todos los individuos que aparecen en una secuencia de videovigilancia, pero que no eran correctamente detectados con el algoritmo de base debido a oclusiones, distintas posturas (como la de sentado) y perspectivas (demasiado cerca de la cámara, y por tanto sin visibilidad del cuerpo entero), pero que innegablemente aparecen en la imagen y deben ser detectadas.

El nuevo detector de personas se ha diseñado utilizando la técnica de ventana deslizante con descriptores HOG (Histograma de Gradientes Orientados o *Histogram of Oriented Gradients*) y un clasificador SVM (Máquina de Soporte Vectorial o *Support Vector Machine*) lineal, con objeto de seguir la misma filosofía del algoritmo implementado en el TFG de partida. De este modo, el nuevo algoritmo resuelve las dos situaciones de detección analizadas: medio cuerpo o cuerpo completo, pues la primera incluye a la segunda (la imagen del cuerpo completo de una persona contiene la de la parte superior de su cuerpo).

En el proyecto se han analizado y puesto en marcha dos versiones diferentes del detector así descrito: una basada en las librerías de *OpenCV*, y otra basada en las librerías de *Matlab*, ambas ampliamente usadas por la comunidad científica para abordar esta tarea concreta de tratamiento de imágenes de *"people detection"*.

Los resultados obtenidos, sin embargo, distan de alcanzar la fiabilidad esperada, concluyendo el proyecto con la propuesta de la investigación y el análisis de nuevas técnicas basadas en otros descriptores de la imagen distintos a HOG para el desarrollo del detector perseguido en futuros trabajos.

Palabras clave: detección y seguimiento, media persona, ventana deslizante, descriptor HOG, clasificador SVM lineal.

Abstract

The objective of this project is the improvement of the people detection and tracking algorithm in 2D colour images from video surveillance scenes, already implemented by Marcos Baptista Ríos in his End of Degree Project [3].

This improvement is based on including the detection of the upper half of the people, needed to identify every individual that appears in a video surveillance sequence, but were not correctly detected by the first algorithm due to occlusions, different postures (like sitting) and perspectives (too close to the camera, and therefore without full body visibility), but undeniably appear in the image and must be detected.

This new people detector has been designed using the sliding window technique with HOG descriptors (Histogram of Oriented Gradients) and a linear SVM classifier (Support Vector Machine), in order to follow the same philosophy of the algorithm implemented in the previous Project. Thereby, this new algorithm resolves the two detection situations analysed: upper half body or full body, because the first one includes the second (a person's full body image contains the upper half body too).

In this project two different versions of the described algorithm were analysed and started: one based on OpenCV libraries, and the other one based on Matlab libraries, both of them widely used by the scientific community to tackle this specific task of image processing for people detection.

The obtained results, however, are far to achieve the expected reliability, closing the project with the proposal for research and analysis of new techniques based on other image descriptors different from HOG for the development of the pursued detector in future works.

Keywords: detection and tracking, upper half body, sliding window, HOG descriptor, linear SVM classifier.

Índice general

Índice de figuras

Índice de tablas

Índice de fragmentos de código fuente

Capítulo 1

Introducción al Trabajo Fin de Grado

La mayoría de la gente vive -ya sea física, intelectual o moralmente- en un círculo muy restringido de sus posibilidades. Todos nosotros tenemos reservas de vida en las que ni siquiera soñamos.

William James

1.1 Presentación

En la actualidad, el reconocimiento de personas y las actividades básicas que realizan se ha convertido en un objetivo fundamental en el campo de la visión artificial. Un sistema capaz de detectar personas y sus comportamientos, capaz de seguir a los individuos detectados y extraer características de la secuencia que se está analizando tiene importantes ventajas en el campo de la videovigilancia y la seguridad. Mediante el uso de sensores como cámaras de color y profundidad, los sistemas de visión artificial permiten identificar personas, controlar aforos, detectar comportamientos extraños, evitar accidentes, etc. Implementar este tipo de sistemas, conseguir buenos resultados en cuanto a la extracción de distintas características, y mejorar el rendimiento en entornos reales y en tiempo real, son las bases de muchos estudios y avances en el ámbito de la visión artificial.

Siguiendo esta línea, el trabajo aquí propuesto continua uno previo realizado dentro del Grupo de Ingeniería Electrónica Aplicada a Espacios Inteligentes y Transporte (GEINTRA) por Marcos Baptista Ríos como trabajo fin de grado [3]. El objetivo es poner en marcha un sistema de detección y seguimiento de personas, capaz de localizar y seguir a todos los individuos que aparezcan en una secuencia de vídeo en color. La mejora que se ha tratado de implementar en el trabajo fin de grado aquí expuesto, respecto al trabajo previo, trata de lo siguiente: solventar los problemas de detección generados por el punto de vista de la imagen, las oclusiones, o las distintas disposiciones de las personas en la imagen mediante la inclusión de un sistema de detección de la mitad superior de las personas ejecutado en paralelo.

Una vez obtenido un sistema funcional y robusto, éste puede servir como base para la implementación e inclusión de algoritmos más complejos que permitan extraer características de comportamiento humano de medio y alto nivel.

1.2 Objetivos

El objetivo global del proyecto es la extracción de características en secuencias de imágenes capturadas con una cámara de color en espacios interiores con el fin de determinar la presencia y cantidad de personas, así como seguir su movimiento durante la secuencia. El objetivo concreto de este trabajo es solventar el problema más importante que se daba en el trabajo de referencia, es decir, resolver mediante un sistema de detección de la mitad superior de las personas los problemas que se originan cuando las personas quedaban ocluidas parcialmente durante la secuencia. Para ello se deben completar ciertos objetivos específicos que complementan y mejoran el trabajo de referencia [3]:

- **1.-** Adecuar una base de datos de imágenes para la detección de la mitad superior de una persona.
- **2-** Utilizar la nueva base de datos para entrenar un clasificador.
- **3.-** Implementar un algoritmo de detección haciendo uso del nuevo clasificador entrenado.

Mediante la extracción de caraterísticas HOG, el uso de un clasificador SVM y el método de ventana deslizante, también utilizado en [3], se puede obtener un sistema de detección que, combinado con el sistema del trabajo de referencia, resuelva los problemas de oclusión mencionados. En la figura 1.1 se muestra un diagrama de bloques que representa las distintas fases en las que se desarrolla este proyecto.

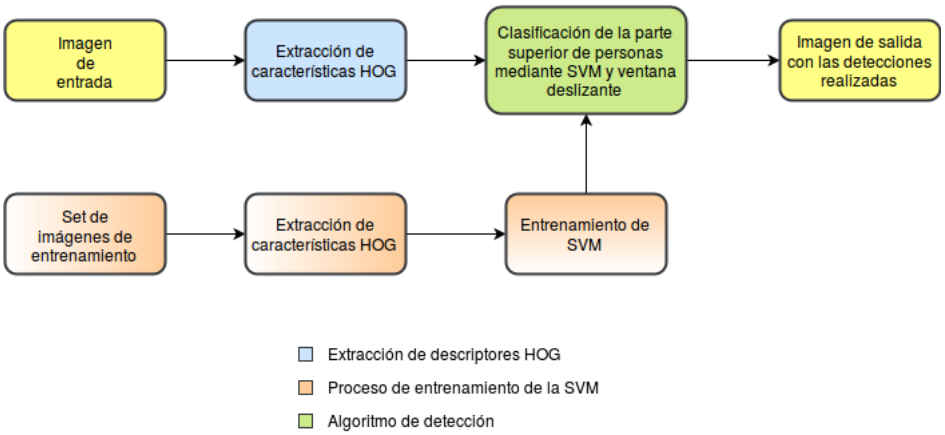

Figura 1.1: Esquema simplificado del proyecto desarrollado.

1.3 Solución propuesta

La solución que se propone en este trabajo para conseguir los objetivos descritos anteriormente se basa en implementar un nuevo sistema con las mismas etapas que el sistema de detección de [3], pero adecuada cada una de ellas a las necesidades de la aplicación.

Por tanto, se ha creado un nuevo modelo de SVM entrenada para detectar la mitad superior de las personas. Para ello fue necesario elaborar un nuevo set de imágenes de entrenamiento a partir del usado en [3], y extraer los descriptores HOG de cada una de las imágenes.

En el trabajo de referencia se hacía uso de una SVM incluida en la librería de *OpenCV* ya entrenada para la detección de personas completas. Sin embargo, en este trabajo se ha realizado de nuevo el proceso de entrenamiento de la SVM con las características que tratamos de detectar.

Del mismo modo, se adecuó el algoritmo de detección utilizado en el anterior trabajo para obtener los resultados objetivo de nuestra aplicación.

Por último, y una vez conseguidos resultados satisfactorios en la detección de la mitad superior de las personas, se debe buscar la forma de integrar los dos sistemas (el del anterior trabajo y éste) para que se ejecuten en paralelo y se resuelvan así los problemas que generan los puntos de vista de las cámaras y las oclusiones parciales de personas que se dan en las secuencias de imagen, objetivo principal de este proyecto.

1.4 Organización de la Memoria

Esta memoria de Trabajo Fin de Grado queda estructurada en 4 secciones que se describen brevemente a continuación.

En el capítulo 1 se introduce y contextualiza el trabajo. Se presentan los objetivos a cumplir y la solución propuesta para su consecución.

En el capítulo 2 se describe la teoría sobre la que se apoya el trabajo desarrollado. Se define brevemente el concepto de espacio inteligente (sección 2.1) así como el proceso de detección de personas (sección 2.2). A continuación se ofrece una explicación sobre los descriptores de características HOG (sección 2.3) para dar paso a un amplio estudio de la teoría de clasificación (sección 2.4). El estudio se divide en la forma de aprendizaje de los clasificadores: Supervisado (2.4.1), No Supervisado (2.4.2) y Semisupervisado (2.4.3).

En el capítulo 3 se explica como se ha implementado el sistema. Una pequeña introducción (sección 3.1) da paso al desarrollo de cada fase del proyecto: extracción de desriptores HOG (sección 3.2), entrenamiento de la SVM (sección 3.3) e implementación del algoritmo de detección (sección 3.4). Posteriormente se comentan las pruebas realizadas, las modificaciones que fueron necesarias y los resultados que ofrecía el sistema (sección 3.5). Como se explica en esta última sección, los resultados con la implementación en *OpenCV* no eran satisfactorios y se procedió a la implementación en *Matlab*, que queda explicada en la sección 3.6, así como se muestran los resultados obtenidos.

Por último, en el capítulo 4 se presentan las conclusiones finales extraídas del desarrollo del proyecto (sección 4.1), así como se proponen distintas líneas futuras que podría tomar el trabajo aquí expuesto (sección 4.2).

Capítulo 2

Estudio teórico

Los sueños de los grandes soñadores jamás llegan a cumplirse, siempre son superados.

Alfred Lord Whitehead

En este capítulo se procede a realizar un estudio de las bases teóricas que apoyan el trabajo realizado. Una breve descripción de los espacios inteligentes y un boceto de cómo funcionan los sistemas de detección de personas da paso al estudio de los descriptores de características HOG. Posteriormente, se describen en profundidad las distintas teorías y modelos de clasificación, organizadas según el tipo de aprendizaje en que se basan.

2.1 Espacio Inteligente

Un entorno inteligente es, de acuerdo con Mark Weiser [4], "un mundo físico que está rícamente entretejido con sensores, actuadores, visualizadores y elementos computacionales, integrados a la perfección en los objetos cotidianos de nuestras vidas, conectados a través de una red continua". Dotar cierto entorno de interfaces hombre-máquina, sensores, actuadores y demás dispositivos electrónicos es una actividad muy común en cualquier ámbito de la ingeniería moderna. Estos espacios inteligentes son capaces de recopilar información del entorno que les rodea, interactuar con el usuario, y llevar a cabo acciones de mayor o menor complejidad; con lo que la realización de gran variedad de tareas se ve enormemente facilitada. Debido a la versatilidad de este tipo de entornos a la hora de diseñarlos e implantarlos, se han convertido en un recurso ampliamente utilizado en numerosos ámbitos de la robótica, tanto industrial como de servicios, en las tareas de seguridad y vigilancia automática, la ayuda a personas con discapacidad, etc. En este tipo de espacios, la visión artificial cobra especial importancia ya que puede ofrecer valiosa información, tanto a los usuarios del sistema, como a los demás subsistemas electrónicos que interactúan. En la figura 2.1 se ve representada la estructura general de un espacio inteligente.

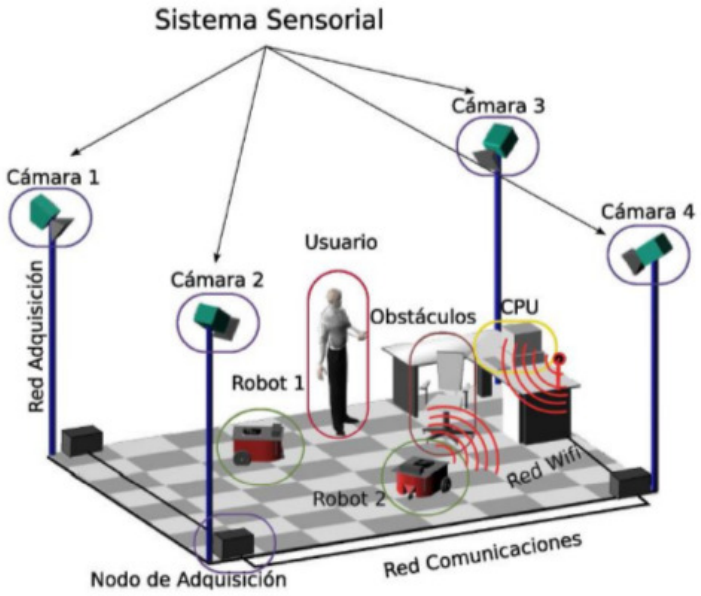

Figura 2.1: Estructura de un espacio inteligente.

2.2 Detección de Personas

El detector de personas visual, es el sistema que se encarga de determinar si aparecen o no personas en una imagen. En caso afirmativo, el detector debe proporcionar información sobre ellas, como el tamaño del individuo o la posición que ocupa en el plano. Este tipo de sistemas se componen de dos módulos o fases: en primer lugar, la extracción de ciertas características de la imagen; y en segundo lugar, la clasificación de las imágenes según algunas de estas características. En los siguientes apartados se explican tanto los descriptores elegidos, como los diferentes clasificadores que pueden emplearse en tareas de detección.

La siguiente sección se centrará en el estudio de los descriptores HOG [5] utilizados en la etapa de extracción de características de las imágenes. La utilización de este tipo de descriptor viene justificado por ser el método que a nivel más básico ha demostrado ofrecer mejores resultados y sirve actualmente como referente para los recientes sistemas de detección. En [3] podemos encontrar un estudio más amplio de las técnicas de extracción de características de una imagen.

2.3 Descriptores de características: HOG

Para abordar cualquier tarea de detección, se necesita una descripción adecuada del objeto a detectar. Para conseguir una descripción fiable se utilizan características que definan por completo al objeto. Estas características se agrupan en un vector típicamente conocido como descriptor de características. Hay que tener en cuenta que diferentes características serán adecuadas para diferentes aplicaciones. La clave está en hacer una elección adecuada de las características: serán mejores aquellas que acentúen las diferencias entre clases y que eliminen las diferencias dentro de una misma clase. Los descriptores basados en histogramas de orientación han demostrado ser especialmente útiles en la eficiente determinación de las características de forma de una imagen. Al ser técnicas que no tienen en cuenta los valores absolutos de intensidad, les permite una mayor invarianza a brillos, sombras y demás cambios de iluminación. Presentan también un gran nivel de detalle en bordes y texturas, al mismo tiempo que permiten cambios moderados de posición.

El descriptor HOG (Histograma de Gradientes Orientados) fue desarrollado por Navneet Dalal [5] basándose en la siguiente idea: la apariencia local de una imagen puede ser caracterizada de forma adecuada por una distribución de gradientes o direcciones de las aristas de una región de la imagen, incluso con un conocimiento pobre de su posición. Tomando ésto como punto de partida, desarrolló un tipo de descriptor consistente en histogramas locales de orientación de los gradientes de regiones distribuidas y solapadas de una imagen, a la cual se le denomina ventana de detección.

El proceso para la extracción de este tipo de detectores se basa en evaluar un conjunto normalizado de histogramas locales de orientación de los gradientes. A continuación, una breve descripción de las etapas que componen este algoritmo:

1.-Normalización color/gamma: consiste en la aplicación de una ecualización de la imagen local previa al resto de procesos, con el objetivo de reducir la influencia de la iluminación. Este paso es opcional ya que no siempre proporciona mejoras a la hora de utilizar estos descriptores en tareas de detección.

2.-Cálculo de gradientes: se trata del paso fundamental a la hora de obtener mejores o peores resultados en la clasficación basada en descriptores HOG. Tiene como objetivo capturar la información del entorno y silueta de la imagen mediante el cálculo de los gradientes de primer o segundo orden píxel a píxel. Como paso previo se pueden aplicar cierto tipo de filtros, aunque la realización más efectiva ha demostrado ser la más básica. Esto se debe a que la información de la imagen se obtiene principalmente de las aristas, y éstas quedan peor definidas al aplicar métodos más elaborados.

3.-Codificación de los histogramas locales de orientación: la formación de los histogramas locales constituye el paso no lineal fundamental de la extracción del descriptor HOG. La codificación propuesta para las características es localmente sensible al contenido de la imagen, aunque tolera pequeños cambios de pose o apariencia. La ventana de detección se divide en pequeñas regiones espaciales, llamadas celdas. Para cada celda, se acumula un histograma unidimensional de orientación de los gradientes de todos los píxeles contenidos en dicha celda. Este histograma a nivel de celda es la representación básica del descriptor HOG.

4.-Normalización del bloque: la intensidad de los gradientes tiene un amplio rango de variación debido a los cambios locales de iluminación y al contraste entre el objeto y el fondo. La etapa de normalización del bloque permite mejorar la invarianza a la iluminación y al sombreado, así como el contraste de los bordes. Como paso inicial en la normalización, se calcula la energía local acumulada en un grupo de celdas al que se denomina bloque. Con esta energía se normaliza cada celda del bloque. Ya que las celdas son compartidas por varios bloques, y que la normalización de cada celda depende del bloque de pertenencia, el resultado difiere al calcularlo de un bloque a otro. Este uso de información redundante en apariencia, no hace si no mejorar el rendimiento del sistema.

5.-Agrupación de los descriptores: el paso final consiste en concatenar los descriptores HOG de todos los bloques, que generalmente se superponen entre sí mediante una rejilla que cubre toda la ventana de detección. El descriptor final contendrá una descripción completa de la ventana de detección y será utlizado en el clasificador posteriormente. Algo a tener en cuenta es que existirán tantas variantes de descriptores HOG como formas haya de seleccionar los bloques a normalizar, pero todas seguirán el mismo algoritmo explicado anteriormente (figura 2.2).

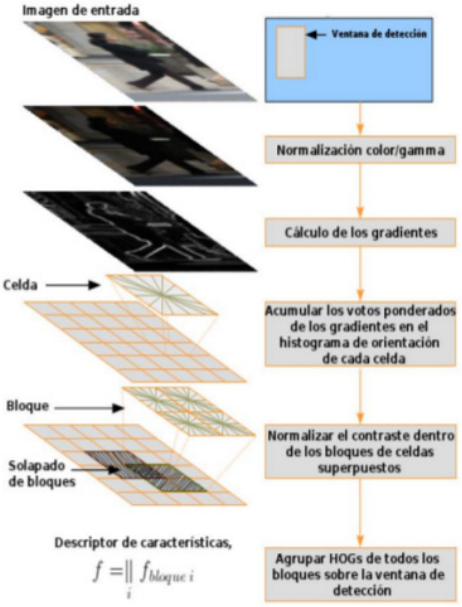

Figura 2.2: Estructura del algoritmo para la extracción de descriptores HOG.

2.4 Clasificadores

La función del clasificador dentro de un sistema de detección cualquiera es la de usar el vector de características proporcionado por el descriptor de características para asignar los objetos detectados a una clase [6]. Debido a que la clasificación directa y perfecta de los objetos es prácticamente imposible, generalmente se recurre a determinar la probabilidad de que cada objeto pertenezca a una posible clase. La cantidad y variedad de datos dada por los vectores de características que representan los objetos de entrada al clasificador ha permitido el desarrollo de un extenso campo denominado teoría de la clasificación.

El grado de dificultad del problema de la clasificación depende de la variabilidad entre los valores dados por el descriptor para las características de objetos dentro de la misma clase frente a la diferencia de esos valores entre las distintas clases. Dentro de la misma clase, esta diferencia de valores puede deberse a la complejidad a la hora de definir la clase o a ruidos, entendiendo ruido en términos generales, es decir: cualquier propiedad del objeto detectado que no se ajusta al modelo planteado ya sea por características aleatorias propias de dicho objeto, o introducidas por los sensores del sistema (cámaras en este caso).

Otro problema que se puede encontrar en la práctica es que puede no ser posible determinar los valores de todas las características de un objeto en particular. Un problema muy común en el campo de la visión artificial son las oclusiones, por ejemplo, que no permiten analizar el objeto detectado por completo, por lo que se pueden perder características claves para la clasificación. Por tanto, la teoría de la clasificación también se centra en como el clasificador debe resolver el problema de la falta de datos para poder determinar óptimamente la clase de la que proviene el objeto en cuestión.

Como se puede deducir, un componente muy importante de la clasificación es la buena elección de las características de las clases que sirven de referencia. En esta tarea, el conocimiento previo de la funcionalidad y las necesidades del sistema cobra una importancia crucial. Sin embargo, la incorporación de estos conocimientos puede ser complicada. En definitiva, a la hora de elegir las características para clasificar los objetos, se debe buscar que éstas sean fáciles de extraer, invariantes frente a pequeñas

transformaciones, poco sensibles al ruido y, obviamente, que definan lo mejor posible la clase en cuestión.

En general, el proceso de usar datos previamente extraídos para definir un clasificador se conoce como entrenar o entrenamiento del clasificador. Anteriormente se han mencionado algunos de los muchos problemas que conlleva el diseño de un clasificador y, como es lógico, no se ha encontrado un método universal que acabe con todos ellos. Sin embargo, los avances en este campo a lo largo de los últimos años han revelado que los métodos de clasificación más efectivos incluyen una importante parte de entrenamiento o aprendizaje a partir de patrones ejemplo.

En conclusión, el diseño de clasificadores pasa por hacer un modelo del clasificador y "enseñarle" mediante ciertos patrones de aprendizaje, para que éste sea capaz posteriomente, de reconocer las características que nos interesan en objetos desconocidos para el clasificador (que no le fueron aportados como tal en el aprendizaje). Debido a la importancia de este entrenamiento, en adelante se va a hacer una distinción de los tipos de clasificadores por cómo realizan este aprendizaje.

2.4.1 Clasificadores basados en Aprendizaje Supervisado

El aprendizaje supervisado es una técnica para deducir una función a partir de datos de entrenamiento. Los datos de entrenamiento consisten en pares de objetos: una componente del par son los datos de entrada y el otro, los resultados deseados. La salida de la función puede ser un valor numérico o la etiqueta de una clase. El objetivo del aprendizaje supervisado es el de crear una función capaz de predecir el valor correspondiente a cualquier objeto de entrada después de haber visto una serie de ejemplos, los datos de entrenamiento. Para ello, tiene que generalizar a partir de los datos presentados a las situaciones no vistas previamente.

2.4.1.1 Clasificadores Bayesianos

Las redes bayesianas, junto con las redes neuronales artificiales, han sido uno de los métodos más utilizados en aprendizaje automático en los últimos años. Es un método importante no sólo porque ofrece un análisis cualitativo de los atributos y valores que puedan intervenir en el problema, sino porque da cuenta también de la importancia cuantitativa de estos atributos. En el aspecto cualitativo podemos representar cómo se relacionan esos atributos ya sea de una forma causal o señalando simplemente la correlación que existe entre esas variables (o atributos). Cuantitativamente (y ésta es la gran aportación de los métodos bayesianos), da una medida probabilística de la importancia de estas variables en el problema, y por tanto, una probabilidad explícita de las hipótesis que se formulan.

Entre las características que poseen los métodos bayesianos en las tareas de aprendizaje, se pueden resaltar las siguientes:

- Cada ejemplo observado va a modificar la probabilidad de que la hipótesis formulada sea correcta. Es decir, una hipóteis que no concuerda con un conjunto de ejemplos más o menos grande no es desechada por completo, sino que lo que harán será aumentar o disminuir la probabilidad estimada para dicha hipótesis.

- Los métodos bayesianos son robustos al posible ruido presente en los ejemplos de entrenamiento y a la posibilidad de tener entre esos ejemplos datos incompletos o posiblemente erróneos.

- Estos métodos permiten tener en cuenta en la predicción de la hipótesis el conocimiento a priori o conocimiento del dominio en forma de probabilidades. El problema puede surgir al tener que estimar ese conocimiento estadístico sin disponer de datos suficientes.

El paradigma clasificatorio en el que se utiliza el teorema de Bayes, en conjunción con la hipótesis de independencia condicional de las variables predictoras dada la clase, se conoce bajo diversos nombres: idiota Bayes [7], Naïve Bayes [8], simple Bayes [9] y Bayes independiente [10]. A pesar de una larga tradición en la comunidad de reconocimiento de patrones, el clasificador Naïve Bayes aparece por primera vez en la literatura del aprendizaje automático a finales de los ochenta [11] con el objetivo de comparar su capacidad predictiva con la de métodos más sofisticados. De manera gradual, los investigadores de esta comunidad de aprendiaje automático han ido dando cuenta de su potencialidad y robustez en problemas de clasificación supervisada. El algoritmo de Naïve Bayes debe su nombre a las hipótesis tan simplificadoras sobre las que se construye dicho clasificador. Dado un ejemplo x representado por k valores, el clasificador Naïve Bayes se basa en encontrar la hipótesis más probable que describa a ese ejemplo. Si la descripción de ese ejemplo viene dada por los valores $[a_1, a_2, .., a_n]$, la hipótesis más probable será aquella que cumpla:

$$v_{MAP} = argmax_{v_j \in V} P(v_j | a_1, ...a_n) \tag{2.1}$$

Es decir, la probabilidad de que, conocidos los valores que decriben a ese ejemplo, éste pertenezca a la clase v_j, donde v_j es el valor de la función de clasificación $f(x)$ en el conjunto infinito V. Por el teorema de Bayes:

$$v_{MAP} = argmax_{v_j \in V} \frac{P(a_1, ...a_n | v_j) p(v_j)}{P(a_1, ...a_n)} = argmax_{v_j \in V} P(a_1, ...a_n | v_j) p(v_j) \tag{2.2}$$

Podemos estimar $P(v_j)$ contando las veces que aparece el ejemplo v_j en el conjunto de entrenamiento y dividiéndolo por el número total de ejemplos que forman este conjunto. Para estimar el término $P(a_1, ...a_n | v_j)$, es decir, las veces en que para cada categoría aparecen los valores del ejemplo x, se debe recorrer todo el conjunto de entrenamiento. Éste cálculo resulta impracticable para un número suficientemente grande de ejemplos por lo que se hace necesario simplificar la expresión anterior. Para ello se recurre a la hipótesis de independencia condicional con el objeto de poder factorizar la probabilidad. Esta hipótesis dice lo siguiente:

Los valores a_j que describen un atributo de un ejemplo cualquiera x son independientes entre sí conocido el valor de la categoría a la que pertenecen.

Así, la probabilidad de observar la conjunción de atributos a_j dada una categoría a la que pertenecen es justamente el producto de las probabilidades de cada valor por separado:

$$P(a_1, ...a_n | v_j) = \prod_i P(a_i | v_j) \tag{2.3}$$

En resumen, el clasificador Naïve Bayes ofrece amplias ventajas desde reducidos tiempos de clasificación y aprendizaje, así como bajos requerimientos de memoria, ofreciendo buenos resultados en muchos dominios. Sin embargo, algunas de sus limitaciones han motivado la investigación de los modelos bayesianos, consiguiendo perfeccionar sus características y los resultados que ofrece. Algunos de estos clasificadores son: semi-Naïve Bayes, Naíve Bayes aumentado a Árbol (TAN), clasificadores bayesianos k Dependientes, redes bayesianas... Un estudio más extenso de la teoría de la clasificación mediante modelos bayesianos se puede encontrar en [6].

2.4.1.2 Redes Neuronales

Desde hace décadas, una amplia parte de los estudios en materia de computación se han centrado en el cerebro y la posibilidad de modelar su comportamiento matemáticamente [12].

La razón es sencilla, el cerebro humano es capaz de procesar a gran velocidad grandes cantidades de información procedentes de los sentidos, combinarla, compararla con información previamente almacenada y procesarla para dar las respuestas adecuadas. Estos modelos se han basado en los esudios de las características principales de las neuronas y sus conexiones y, aunque estos modelos son aproximaciones muy lejanas de estas células biológicas, son muy interesantes por su capacidad de aprender y asociar patrones. Con las redes neuronales, se busca la solución a problemas complejos, no como una secuencia de pasos, sino como la evolución de unos sistemas de computación inspirados en el cerebro humano, y dotados, por tanto, de cierta inteligencia. Estos sistemas han sido especialmente útiles en aplicaciones de reconocimiento de formas y patrones, predicción, codificación, control y optimización de procesos entre otras.

Alan Turing, en 1936, fue el primero en estudiar el cerebro como una forma de ver el mundo de la computación, pero quienes primero concibieron algunos fundamentos de la computación neuronal fueron Warren McCulloch y Walter Pitts. Más adelante, Donald Hebb postuló otras teorías, y ya en 1957, Frank Rosenblatt comenzó con el desarrollo del Perceptrón, la red neuronal más antigua. Posteriormente apareció el modelo ADALINE, desarrollado por Bernard Widrow y Marcial Hoff. Investigaciones más recientes, como las de Kuhiniko Fkushimika y Teuvo Kohonen se centraron en el uso de redes neuronales para el reconocimiento de patrones, o el desarrollo del Asociador Lineal por parte de James Anderson.

Como se ha comentado, las redes neuronales artificiales son modelos que intentan reproducir el comportamiento del cerebro, y como tal, realizan una simplificación, averiguando cuáles son los elementos relevantes del sistema. [1] Los elementos característicos de una red neuronal son los siguientes:

- **Neurona Artificial:** se trata de la unidad fundamental de proceso, y se distinguen entre ellas tres tipos: de entrada, de salida y ocultas. Las unidades de entrada se encargan de recibir las señales del entorno. Las de salida envían las señales fuera de la red. Por último, las unidades ocultas son aquellas cuya entrada y salida se encuentra dentro del sistema, y se encargan de procesar la información y comunicarla a otras capas. Cada una de estas unidades de proceso se compone a su vez de una red de conexiones de entrada; una función de red, o función de propagación; un núcleo central de proceso, encargado de aplicar la función de activación; y la salida, que transmite el valor de activación a otras unidades (2.3).

  - **Función de propagación:** calcula el valor de entrada total a la unidad, normalmente mediante la suma de todas las entradas recibidas ponderadas por el peso o valor de las conexiones.

  - **Función de activación:** consiste en la característica clave de las neuronas, la que define su comportamiento. Se encarga de calcular el nivel o estado de activación de la neurona en función de la entrada. Se pueden utilizar distintos tipos de funciones, desde funciones umbral simples hasta funciones no lineales.

  - **Conexiones ponderadas:** recrean el papel de las conexiones sinápticas del cerebro. La existencia de conexiones determina si es posible que una unidad influya sobre otra; el valor de los pesos y su signo definen el tipo (excitatorio o inhibitorio) y la intensidad de esa influencia.

  - **Salida:** calcula la salida de la neurona en función de la activación de la misma. El valor de salida cumpliría la función de la tasa de disparo de las neuronas biológicas.

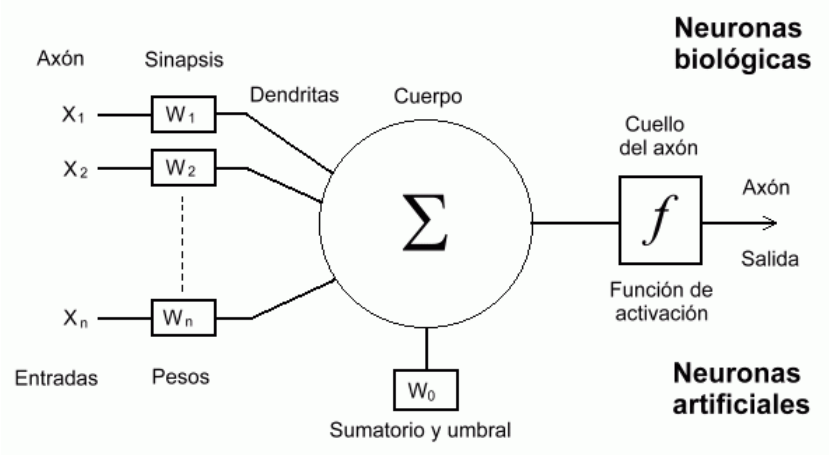

Figura 2.3: Neurona artificial: elementos y símil con la neurona biológica [1].

- **Estructura de la red neuronal:** Las redes neuronales están formadas típicamente por una serie de capas de neuronas unidas entre sí. Cabe definir el concepto capa o nivel como conjunto de neuronas cuyas entradas provienen de la misma fuente y cuyas salidas se dirigen al mismo destino. Podemos distinguir varias estructuras según la conexión entre las neuronas de la red:
  - **Unión todos con todos:** cada neurona de una capa se une a todas las neuronas de la siguiente capa. Es el tipo de conexionado más utilizado en redes neuronales, desde el Perceptrón multicapa a las redes de Hopfield.
  - **Unión lineal:** consiste en unir cada neurona de una capa con una única neurona de la capa siguiente. Se suele utilizar para unir la capa de entrada con la capa de procesamiento cuando la primera se utiliza como sensor.
  - **Predeterminado:** es un tipo de conexionado que permite a las redes agregar o eliminar neuronas de sus capas, y por tanto eliminar ciertas conexiones.

  Estableciendo un orden en las capas de la red, podemos definir conexiones hacia delante, hacia atrás y laterales. Esto nos proporciona otra clasificación de las redes en: *feedforward*, que no tienen conexiones hacia detrás; y *feedback*, que sí permiten este tipo de conexionado. Las conexiones laterales hacen referencia a conexiones de neuronas de la misma capa, muy comunes en redes monocapa. Las redes neuronales también pueden permitir que las neuronas tengan conexiones a sí mismas, dando a la red el nombre de autorecurrente.

Una vez definidos los elementos característicos de una red neuronal, podemos proceder a clasificar los tipos de redes neuronales. Esta clasificación viene dada por dos propiedades de las redes: topología de la red y método de aprendizaje.

La arquitectura de las redes neuronales consiste en la organización y disposición de las neuronas formando capas más o menos alejadas de la entrada y la salida del sistema. En este sentido, los parámetros fundamentales de la red son: número de capas, número de neuronas por capa, grado de conectividad y tipo de conexiones (mencionadas anteriormente). Podemos diferenciar entonces entre redes monocapa y redes multicapa:

- **Redes monocapa:** las conexiones que se establecen son laterales o autorecurrentes, ya que existe una única capa en la red. Se utilizan en tareas relacionadas con lo que se conoce como autoasociación, es decir, generar informaciones de entrada que se presentan a la red de forma incompleta o distorsionada.

- **Redes multicapa:** poseen conjuntos de neuronas agrupados en distintas capas. Para distinguir la capa a la que pertenece una neurona, basta con fijarse en el origen de las señales de entrada que recibe y el destino de la señal de salida.

El aprendizaje en una red neuronal se basa en modificar los pesos de las conexiones en función de la entrada. Estos cambios se reducen a la destrucción, modificación y creación de conexiones entre las neuronas. La creación de una nueva conexión implica que el peso de la misma pasa a ser un valor distinto de 0. El proceso de aprendizaje se puede dar por finalizado cuando los valores de todos los pesos permanecen estables. Según este aprendizaje, podemos clasificar las redes neuronales entre redes de aprendizaje supervisado y de aprendizaje no supervisado, que trateremos más adelante [2.4.2.1].

Teniendo en cuenta la base sobre la que se apoya el aprendizaje supervisado [2.4.1], se pueden considerar tres formas de llevar a cabo este aprendizaje:

- **Aprendizaje por corrección:** consiste en ajustar los pesos en función de la diferencia entre los valores deseados y los valores obtenidos en la salida de la red.
- **Aprendizaje por refuerzo:** se basa en la idea de no indicar durante el entrenamiento el valor exacto de la salida que se desea que proporcione la red ante una determinada entrada. La función del supervisor se reduce a indicar mediante una señal de refuerzo si la salida obtenida en la red se ajusta a la deseada (éxito = +1 o fracaso = -1), y en función de ello se ajustan los pesos basándose en un mecanismo de probabilidades.
- **Aprendizaje estocástico:** este tipo de aprendizaje consiste básicamente en realizar cambios aleatorios en los pesos de las conexiones de la red y evaluar su efecto a partir del objetivo deseado y de distribuciones de probabilidad.

Las redes más significativas que utilizan el aprendizaje supervisado son el Perceptrón, el Perceptrón multicapa y la red de Hopfield.

Perceptrón

El perceptrón es un clasificador que asigna a un vector de N valores un valor binario utilizando una transformación no lineal. Así, cada vector pertenece a una de las particiones que crea el perceptrón. Esta red es una máquina de computación universal y tiene la expresividad equivalente a la lógica binaria, ya que podemos crear un perceptrón con el mismo comportamiento que una función booleana NAND y a partir de esta función se puede crear cualquiera de las otras funciones booleanas.

En cuanto a la arquitectura de esta red, ésta consta de dos capas de neuronas. Admite valores binarios como entrada para los sensores y los valores a la salida se encuentran en el mismo rango que los de entrada. La primera capa hace de sensor, la segunda realiza todo el procesamiento y las neuronas de la red se conexionan todas con todas.

Perceptrón multicapa

El perceptrón simple tiene una serie de limitaciones importantes. La más relevante es que no tiene capacidad para clasificar conjuntos que no son linealmente independientes. En 1969 se demostró que un perceptrón es incapaz de aprender la función XOR. El modelo multicapa es una ampliación del perceptrón

que, como su nombre indica, añade una serie de capas que se encargarán de realizar transformaciones sobre las variables de entrada. Estas transformaciones se orientan a convertir funciones linealmente dependientes en independientes, acabando con el problema del perceptrón simple. Gracias a esto y a que el perceptrón multicapa admite valores reales, podemos decir que es un modelador de funciones universal 2.4.

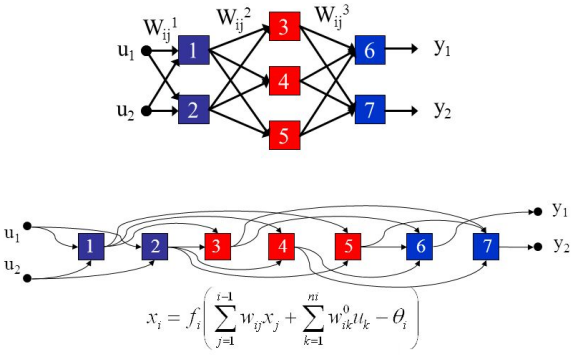

Figura 2.4: Perceptrón Multicapa: Estructura general y expresión general de la función de la red [1].

Red de Hopfield

La red de Hopfield es una de las redes neuronales más importantes y ha influido en el desarrollo de multitud de redes posteriores. Se trata de una red autorecurrente en la que, las conexiones entre las neuronas de la primera y la segunda capa son lineales, mientras que en la segunda capa las neuronas se conectan todas con todas. El hecho de que todas las neuronas de la capa media se encuentren interconectadas hace que en esta capa se dé un feedback entre ellas. De tal forma que, al activarse una neurona de esta capa hace que las otras neuronas cambien su estado de activación, que a la vez hará cambiar el de la primera 2.5.

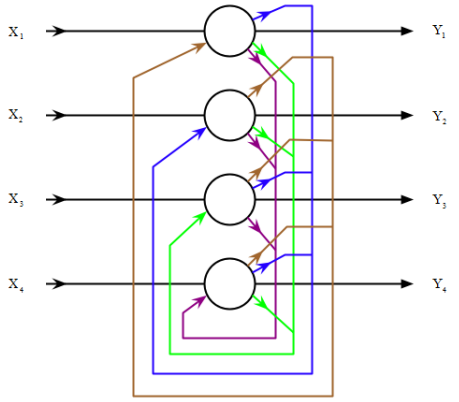

Figura 2.5: Red de Hopfield: Estructura general de la red [1].

2.4.1.3 Clasificadores basados en el Análisis de Componentes Principales

En estadística, el PCA, o Análisis de Componentes Principales, es una técnica utilizada para reducir la dimensionalidad de un conjunto de datos. Técnicamente, el PCA busca la proyección según la cual los datos quedan mejor representados en términos de mínimos cuadrados. Se utiliza sobre todo en el análisis exploratorio de datos y para construir modelos predictivos.

Existen diferentes técnicas para la reducción de la dimensionalidad, basadas en la realización de una transformación lineal del espacio original a un nuevo espacio transformado. PCA se distingue entre estas técnicas por ser muy apropiada para aplicaciones de clasificación, ya que permite representar óptimamente en un espacio de pequeñas dimensiones observaciones de un espacio multidimensional de gran tamaño. Además, permite transformar las variables originales, por lo general muy correladas, en nuevas variables incorreladas, facilitando en gran medida la interpretación de la información que contienen.

A continuación se analizarán brevemente los fundamentos matemáticos del PCA, para posteriormente, dar una idea de cómo se puede aplicar esta técnica a la clasificación.

PCA es una técnica ampliamente utilizada en estadística, por lo que en la literatura se pueden encontrar numerosas referencias a sus fundamentos matemáticos [2]. Dado que estamos estudiando su utilidad como clasificador, aquí se tratará brevemente su formulación para ser aplicado en un proceso de validación de medidas. Posteriormente, entender su funcionalidad como clasificador se vuelve más intuitivo. Los datos sobre los que se aplica PCA serán el conjunto de medidas tomadas por un sistema, que se agrupan en forma de vector columna y. Este vector se conoce como el vector de medidas, y está asociado al espacio de medidas. Entonces, con PCA realizaremos una transformación lineal de ese espacio de medidas, de dimensión n, a un espacio m-dimensional, con $m \leq n$, mediante la matriz de transformación $U = [u_1\ u_2\ ...\ u_m]_{nxm}$, real y de dimensiones nxm, cuyas columnas son linealmente independientes y forman un espacio ortonormal.

La aplicación de PCA a la validación de medidas conlleva dos fases diferentes [2]. La primera fase se trata de un entrenamiento en el que se obtiene la matriz de transformación U a partir de un conjunto de vectores que se proporciona al sistema (esto hace que consideremos los clasificadores PCA dentro de los clasificadores con aprendizaje supervisado). Este proceso se reduce a: la obtención de los vectores de entrenamiento, estimación de la media y obtención de vectores de media nula, cálculo de la matriz de covarianza, cálculo de autovalores y autovectores, y por último, obtención de la matriz de transformación [2.6].

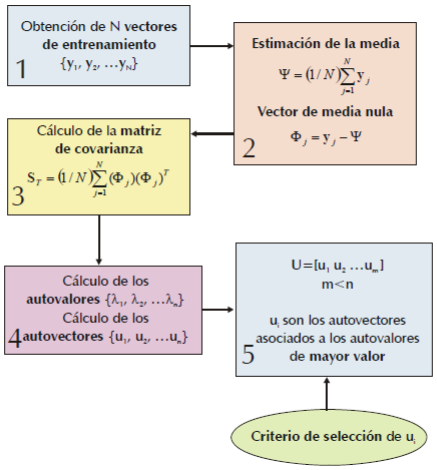

Figura 2.6: Resumen de las operaciones que se deben realizar para obtener la matriz de transformación U. Fase *off-line*[2].

En la segunda fase, dado un vector de medidas, este se transforma al espacio de características mediante la ecuación 2.4, en la que Φ es el vector de medida con media nula que transformaremos, Ω será el vector resultado de esta transformación, y U, la matriz de transformación obtenida durante la fase de entrenamiento:

$$\Omega = U^T \Phi \tag{2.4}$$

Tras la transformación anterior, la recuperación del vector $\hat{\Phi}$ a partir del espacio transformado se realiza a través de la transformación inversa:

$$\hat{\Phi} = U^T \Omega \tag{2.5}$$

La diferencia entre el vector de medidas original Φ y el recuperado $\hat{\Phi}$ se conoce como error o distancia de recuperación:

$$\epsilon_{PCA} = \left\| \Phi - \hat{\Phi} \right\| \tag{2.6}$$

Cuanto mayor es la diferencia entre m y n, el error que se produce en la recuperación es mayor. En caso de que, dado cierto vector de medidas, se cumpla que el error de recuperación de PCA sea menor que un umbral establecido, significará que ese vector tiene gran semejanza con alguno de los vectores de entrenamiento utilizados en la obtención de la matriz de transformación U.

Extrapolando a la clasificación, si el proceso de entrenamiento se realiza, en vez de con vectores que representen medidas, con vectores que representen ciertas características o patrones, el proceso acaba ofreciéndonos la información de cómo de parecidos son los patrones nuevos que presentamos al clasificador respecto a los utilizados en el entrenamiento. Cada uno de los vectores de entrenamiento utilizados representará cada una de las clases que necesitemos en nuestra aplicación, y PCA nos dará la semejanza de los objetos que analiza con cada una de esas clases, siendo clasificado el objeto en la clase a la que más se parezca.

2.4.1.4 SVM

Las máquinas de soporte vectorial (*Support Vector Machines* en inglés, SVM) tienen su origen en los trabajos sobre la teoría del aprendizaje estadístico, introducidas en los años 90 por Vapnik y sus colaboradores [13] [14] . Aunque originalmente fueron pensadas para resolver problemas de clasificación binaria, actualmente se utilizan para resolver diversos tipos de problemas (regresión, agrupamiento, multiclasificación, etc.). Sus características permiten a las SVMs ser aplicadas con éxito en muy diversos campos como la visión artificial, reconocimiento de caracteres, categorización de textos e hipertextos, clasificación de proteínas, procesamiento de lenguaje natural o análisis de series temporales. De hecho, desde su introducción, han ido ganando un merecido reconocimiento gracias a sus sólidos fundamentos teóricos [13].

Dentro de la tarea de clasificación, las SVMs pertenecen a la categoría de clasificadores lineales, puesto que inducen separadores lineales o hiperplanos, ya sea en el espacio original de los ejemplos de entrada, o en un espacio transformado (espacio de características) si los ejemplos no son separables linealmente en el espacio original. La búsqueda de estos hiperplanos de separación en esos espacios transformados (normalmente de muy alta dimensión) se hará de forma implícita utilizando funciones kernel.

La mayoría de métodos de aprendizaje se centran en minimizar los errores cometidos por el modelo creado a partir de los ejemplos de entrenamiento. Sin embargo, la teoría de las SVMs se centra en la minimización de lo que se conoce como riesgo estructural. La idea es elegir un hiperplano de separación mínima que equidiste de los ejemplos más cercanos de cada clase para conseguir así un margen máximo

entre las muestras a cada lado del hiperplano. Además, a la hora de definir el hiperplano, sólo se consideran los ejemplos de entrenamiento de cada clase que caen justo en la frontera de dichos márgenes. Estos ejemplos reciben el nombre de vectores soporte.

Desde el punto de vista práctico, la obtención del hiperplano de margen máximo ha demostrado tener una buena capacidad de clasificación, solventando además gran cantidad de problemas derivados del sobreajuste a los ejemplos de entrenamiento. Desde el punto de vista de la algoritmia, el problema de optimización del margen geométrico representa un problema de optimización cuadrático con restricciones lineales que puede ser abordado mediante técnicas de programación cuadrática estándar. Las propiedades sobre las que se fundamenta la resolución de este tipo de problemas garantizan una solución única, a diferencia, por ejemplo, de las redes neuronales, que entrenadas con un mismo conjunto de ejemplos, pueden encontrar más de una solución.

Refiriéndonos al problema de la clasificación, podemos dividir las máquinas de soporte vectorial en dos grandes grupos: máquinas de soporte vectorial binarias y máquinas de soporte vectorial multiclase. La diferencia reside en la cantidad de clases que son capaces de generalizar. Como el nombre indica, las binarias son capaces de discernir entre dos clases (en nuestro caso, personas y no personas), mientras que las multiclase pueden diferenciar entre mayor variedad de clases.

2.4.1.4.1 SVM para la clasificación binaria

Como ya hemos comentado, este tipo de SVM puede diferenciar entre dos clases. A la máquina se le entrega una serie de ejemplos de entrenamiento con los que definirá el hiperplano de separación entre clases, y posteriormente, será capaz de reconocer y clasificar ejemplos que no se le hayan mostrado antes. Dependiendo de la disposición de las muestras, si éstas pueden ser perfectamente separables, se puede utilizar las SVM *hard margin*, o SVM de margen duro.

Dado un conjunto separable de ejemplos $S = \{(x_1, y_1), ..., (x_n, y_n)\}$, donde $x_i \in \mathbb{R}^d$ e $y_i \in \{+1, -1\}$, se puede definir un hiperplano de separación como una función lineal que es capaz de separar dicho conjunto sin error:

$$D(x) = (w_1 x_1 + ... + w_d x_d) + b =< w, x > +b \tag{2.7}$$

Donde w y b son coeficientes reales. El hiperplano de separación cumple las siguientes restricciones para todo x_i del conjunto de ejemplos:

$$< w, x_i > +b \geq 0 \quad si \quad y_i = +1 \tag{2.8}$$

$$< w, x_i > +b \leq 0 \quad si \quad y_i = -1 \quad i = 1, ..., n \tag{2.9}$$

O, de manera reducida:

$$y_i D(x_i) \geq 0, \quad con \quad D(x_i) =< w, x_i > +b \quad para \quad i = 1, ..., n \tag{2.10}$$

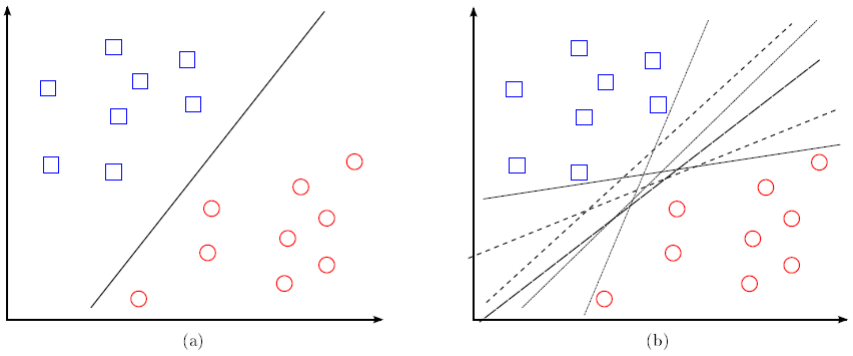

Figura 2.7: Representación de los hiperplanos de separación: en (a) el hiperplano de separación óptimo, en (b) distintos hiperplanos posibles [3].

Según la figura 2.7, se puede deducir que el hiperplano que separa los ejemplos no es único, sino que existen infinitos hiperplanos de separación que son capaces de cumplir las restricciones impuestas por las expresiones anteriores. La búsqueda se centra ahora en encontrar el hiperplano de separación óptimo. Para ello se debe definir algún criterio más que caracterice ese término "óptimo". El concepto margen de un hiperplano ayuda a resolver este problema. Se establece el margen (τ) como la mínima distancia de separación entre un hiperplano y el ejemplo más cercano de cualquiera de las dos clases. A partir de esta definición, un hiperplano de separación se denominará óptimo si su margen es máximo.

Una propiedad inmediata de la definición de hiperplano de separación óptimo es que éste equidista del ejemplo más cercano a cada clase. Mediante geometría, sabiendo que la distancia entre un hiperplano $D(x)$ y un ejemplo $x^{'}$ viene dada por:

$$\frac{\left|D(x^{'})\right|}{\|w\|} \tag{2.11}$$

Siendo $|\cdot|$ el operador valor absoluto, $\|\cdot\|$ el operador norma de un vector y w el vector que junto a b define el hiperplano $D(x)$, y que, además, tiene la propiedad de ser perpendicular al hiperplano considerado. Haciendo uso de las expresiones 2.10 y 2.11, todos los ejemplos de entrenamiento cumplirán que:

$$\frac{y_i D(x_i)}{\|w\|} \geq \tau \quad para \quad i = 1, ..., n \tag{2.12}$$

De esta expresión se deduce que encontrar el hiperplano óptimo es equivalente a encontrar el valor w que maximiza el margen. Sin embargo, existen infinitas soluciones que difieren solo en la escala de w. Para limitar el número de soluciones a una sola, teniendo en cuenta que 2.12 se puede reescribir como:

$$y_i D(x_i) \geq \tau \|w\| \quad para \quad i = 1, ..., n \tag{2.13}$$

Fijamos la escala del producto del término de la derecha a la unidad:

$$\tau \|w\| = 1 \tag{2.14}$$

Llegamos a la conclusión de que aumentar el margen es equivalente a disminuir la norma de w, ya que la expresión anterior equivale a:

$$\tau = \frac{1}{\|w\|} \tag{2.15}$$

Por tanto, de acuerdo con su definición, un hiperplano de separación óptimo [2.7] será aquel que posee un margen máximo, y por tanto, un valor mínimo de $\|w\|$ y, además, esta sujeto a la restricción dada por 2.13 junto con el criterio expresado en 2.14:

$$y_i D(x_i) \geq 1 \quad para \quad i = 1, ..., n \tag{2.16}$$

En conclusión, el concepto de margen máximo está relacionado directamente con la capacidad de generalización del hiperplano de separación, de tal forma que, a mayor margen, mayor distancia de separación existirá entre las dos clases. Los ejemplos que están situados a ambos lados del hiperplano óptimo y que definen el margen (aquellos que cumplen 2.16 como una igualdad) son conocidos como los vectores soporte. Puesto que estos ejemplos son los más cercanos al hiperplano, serán los más difíciles de clasificar, y por tanto, son los únicos a considerar a la hora de construir dicho plano. La búsqueda del hiperplano óptimo para este caso se formaliza como el problema de optimización de encontrar el valor de w y b que minimiza la norma de w sujeto a las restricciones de 2.16:

$$f(w) = \frac{1}{2} \|w\|^2 \tag{2.17}$$

Para resolver este problema, se llega a la función lagrangiana de la ecuación 2.18 empleando multiplicadores de *Lagrange*:

$$L(w, b, \alpha) = \frac{1}{2} w^T w - \sum_{i=1}^{N} \alpha_i [y_i (w^T x_i + b) - 1] \tag{2.18}$$

Desarrollar esta ecuación da lugar a un sistema de ecuaciones resoluble para datos de entrada linealmente separables, obteniendo un mínimo global:

$$\begin{cases} \frac{\partial L}{\partial w} \rightarrow w = \sum_{i=1}^{N} \alpha_i y_i x_i \\ \\ \frac{\partial L}{\partial w} \rightarrow w = \sum_{i=1}^{N} \alpha_i y_i = 0 \end{cases} \tag{2.19}$$

Este sistema da lugar a la ecuación:

$$w^T w = w^T \sum_{i=1}^{N} \alpha_i y_i x_i = \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i x_j \tag{2.20}$$

Sustituyendo en 2.18, se obtiene la función que se debe maximizar como sigue:

$$J(w, b, \alpha) = Q(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i x_j \tag{2.21}$$

Con la restricción $\sum_{i=1}^{N} \alpha_i y_i = 0$ para $\alpha_i \geq 0$, $i = 1, ..., N$ siendo α_i los correspondientes a los vectores soporte, los únicos distintos de cero.

Queda demostrado entonces que el hiperplano de separación óptimo depende únicamente de los vectores soporte extraídos de los ejemplos de entrenamiento. Una vez obtenidos los valores de α_i^*, los coeficientes que definen el hiperplano se obtienen mediante la ecuación 2.22:

$$\alpha_i \Rightarrow w^* = \sum_{i=1}^{N} \alpha_i^* y_i x_i \rightarrow b^* = 1 - w^{*T} x_s \tag{2.22}$$

2.4.1.4.2 SVM de margen relajado

Los problemas que resuelven las SVMs de margen duro comentadas en el apartado anterior resultan tener poca validez práctica en la aplicación real, ya que normalmente los ejemplos reales con los que se realiza el aprendizaje de la SVM están expuestos a ruidos y no suelen ser perfecta y linealmente separables. Este tipo de problemas se afronta relajando el margen de separabilidad del conjunto de ejemplos, permitiendo errores de clasificación en algunos de los ejemplos del conjunto de entrenamiento 2.8. Sin embargo, el objetivo seguirá siendo encontrar el hiperplano óptimo de separación para el resto de ejemplos que sí son separables. Retomando las restricciones de la sección anterior, se establece que un ejemplo es no separable cuando no cumple la condición 2.16. Se pueden dar dos casos: el ejemplo cae dentro del margen asociado a la clase correcta, de acuerdo con el límite de decisión establecido por el hiperplano de separación; o el ejemplo cae al otro lado del hiperplano. En cualquier caso, se dice que el ejemplo es no-separable, pese a que en el primer caso el ejemplo es clasificado correctamente.

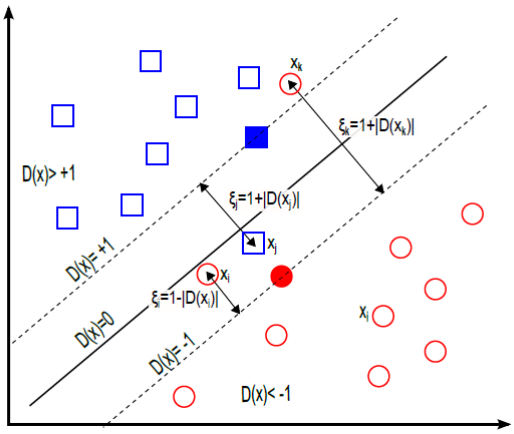

Figura 2.8: Representación de la SVM de margen relajado [3].

La idea para abordar este problema se basa en introducir en la condición 2.16 (que define el hiperplano de separación) un conjunto de variables reales positivas que permiten cuantificar el número de ejemplos no-separables que se está dispuesto a admitir. Estas variables son las denominadas *variables de holgura* (ξ_i):

$$y_i D(x_i) \geq 1 - \xi_i \;\; ; \;\; con \quad \xi_i \geq 0 \;\; ; \;\; D(x_i) =< w, x_i > +b \quad para \quad i = 1, ..., N \tag{2.23}$$

Entonces, para un ejemplo cualquiera (x_i, y_i), su variable del holgura ξ_i representa la desviación del caso separable medida desde el borde del margen correspondiente a la clase 2.8. De acuerdo con esta definición, las variables de holgura de valor nulo corresponden a ejemplos perfectamente separables, valores mayores que 0 corresponden a ejemplos no separables pero bien clasificados, mientras que variables de holgura mayores que 1 representan ejemplos no separables y clasificados de forma errónea. Entonces, la suma de todas las variables de holgura permite medir el coste asociado al número de ejemplos no separables. Cuanto mayor sea el valor de esta suma, mayor la cantidad de ejemplos no separables. Una vez incluidas en el modelo estas variables, ya no basta con maximizar el margen del hiperplano, ya que,

sin tener en cuenta otras necesidades, podríamos encontrar este máximo a costa de clasificar de forma errónea demasiados ejemplos. Por tanto, la función a optimizar debe incluir de alguna manera los errores de clasificación cometidos por el hiperplano de separación:

$$f(w,\xi) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\xi_i \tag{2.24}$$

Donde C es una constante elegida por el usuario, suficientemente grande como para controlar en qué grado influyen los ejemplos no separables en la minimización de la norma. Esta constante permitirá regular el compromiso entre el grado de sobreajuste del clasificador final y la proporción de ejemplos no separables. A mayor C, los valores que se permiten de ξ_i son más pequeños. Del mismo modo, para un valor muy reducido de C se permiten valores muy grandes de ξ_i, permitiendo una gran cantidad de ejemplos mal clasificados. Generalizando, el límite $(C \to \infty)$ representa el caso de ejemplos perfectamente separables $(\xi_i \to 0)$; mientras que en el límite $(C \to 0)$ se permitiría que todos los ejemplos estuvieran mal clasificados $(\xi_i \to \infty)$.

En resumen, el nuevo problema de optimización pasa por encontrar el hiperplano definido por w y b que minimiza 2.24, sujeto a las restricciones 2.23. El hiperplano definido recibe el nombre de hiperplano de separación de margen blando o relajado. El procedimiento para resolver el problema de optimización de este caso es similar al del hiperplano de margen duro. Siguiendo un desarrollo análogo al de la sección anterior, se obtendría que las variables de ajuste desaparecen, quedándose únicamente la constante C como restricción adicional a los multiplicadores de *Lagrange*:

$$L(w,b,\xi,\alpha,\beta) = \frac{1}{2} < w,w > +C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\alpha_i[y_i < w,x_i > +b) + \xi_i - 1] - \sum_{i=1}^{N}\beta_i\xi_i \tag{2.25}$$

Estableciendo las relaciones oportunas:

$$\sum_{i=1}^{N}\alpha_i y_i = 0 \tag{2.26}$$

$$C = \alpha_i + \beta_i \tag{2.27}$$

Obtenemos finalmente la función que debemos maximizar:

$$L(\alpha) = \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j x_i x_j \tag{2.28}$$

Teniendo en cuenta las restricciones:

$$\begin{gathered}\textstyle\sum_{i=1}^{N}\alpha_i y_i = 0 \\ 0 \le \alpha_i \le C,\ \ i = 1,...,N\end{gathered} \tag{2.29}$$

Como en el caso del margen duro, la solución nos permite expresar el hiperplano de separación óptima en términos de α^* y β^*:

$$w^* = \sum_{i=1}^{N} \alpha_i^* y_i < x, x_i > + b^* \beta^* = y_i - < w^*, x_i > \quad \forall i \Rightarrow 0 < \alpha_i < C \tag{2.30}$$

Cabe destacar que en este caso, a diferencia del anterior, para el calculo de β^*, no es suficiente con elegir cualquier ejemplo de x_i que tenga asociado un $\alpha_i > 0$; si no que el ejemplo elegido x_i debe tener asociado un α_i que cumpla la restricción $0 < \alpha_i < C$.

En resumen, en el caso del margen relajado, hay dos tipos de ejemplos para los que $\alpha_i^* \neq 0$. Los que cumplen $0 < \alpha_i^* < C$ que conocemos como vectores soporte; y aquellos para los que $\alpha_i^* = C$, asociados a ejemplos no separables. Estos últimos reciben el nombre de vectores soporte acotados. Ambos tipos de vectores (extraídos de los ejemplos) intervienen en la construcción del hiperplano de separación.

2.4.1.4.3 SVM no lineal

En los apartados anteriores se ha visto que se puede obtener una clasificación muy acertada de clases utilizando ejemplos que son separables linealmente, ya sea directamente, o aceptando pequeños errores, incluso independientemente de la dimensionalidad del problema. Sin embargo, podemos enfrentarnos a problemas en los que una función lineal no permita separar los ejemplos de forma correcta. En estos casos, la solución pasa por transformar el espacio de los ejemplos en uno nuevo, denominado *espacio de características*. La idea es construir un hiperplano de separación lineal en este espacio. Una vez obtenido, se transformará de nuevo al espacio original, resultando una frontera de decisión no lineal.

En la figura 2.9 se puede observar de forma gráfica la idea en la que se basa la SVM no lineal.

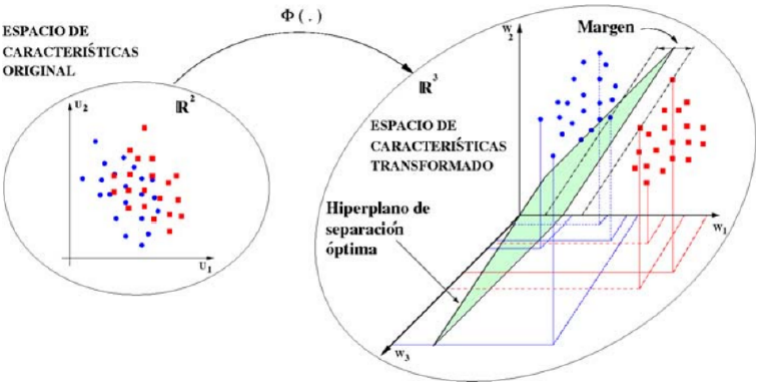

Figura 2.9: Representación general de la clasificación mediante SVM no lineal [3].

Siendo la ecuación 2.31 la expresión matemática de la transformación no lineal $\Phi(\cdot)$ de los datos de entrada a un espacio de dimensión superior, una vez obtenido el nuevo espacio, se puede proceder a la separación lineal de los datos transformados:

$$\mathbb{R}^n \xrightarrow{\Phi} \mathbb{R}^m \,, \quad m > n \tag{2.31}$$

En el espacio de características, la función de decisión (2.7) viene dada por:

$$D(x) = (w_1 \phi_1(x) + ... + w_m \phi_m(x)) =< w, \phi(x) > \tag{2.32}$$

O también:

$$D(x) = \sum_{i=1}^{n} a_i^* y_i K(x, x_i) \tag{2.33}$$

Donde $K(x, x_i)$ se denomina función kernel. Por definición, una función kernel es una función que asigna a cada par de elementos de un espacio de entrada, un valor real correspondiente al producto escalar de las imágenes de dichos elementos en un nuevo espacio (en nuestro caso el espacio de características). Estas funciones nos permiten resolver el problema de la misma forma que en las SVMs de margen relajado (2.28), que sigue siendo encontrar el valor de los parámetros a_i^* para la optimización. El problema queda ahora expresado como:

$$L(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \tag{2.34}$$

Manteniendo las restricciones 2.29.

2.4.1.4.4 SVM multiclase

Cabe destacar que todos los desarrollos matemáticos de esta sección dedicada a SVMs se realizan para la clasificación en dos clases distintas, típico de las SVM tradicionales. Sin embargo, es perfectamente válido el desarrollo de SVMs para la diferenciación de un mayor número de clases. La idea principal se basa en implementar distintas SVMs y combinar de alguna manera los datos que nos ofrecen. Las estrategias más empleadas son:

- **One-against-all:** del inglés, uno contra todos, se diseña una SVM por cada clase, entrenada para distinguir entre dicha clase y todas las demás. La clase de pertenencia se decide por la maximización de las salidas.

- **One-against-one:** del inglés, uno contra uno. El procedimiento se basa en diseñar $C(C-1)/2$ clasificadores binarios, donde C es el número de clases. Cada uno distingue entre cada par de clases y el resultado final se decide mediante votación. En caso de empate, el clasificador de las clases empatadas será el que decida. También se puede concluir esta decisión por el metodo anterior.

2.4.1.5 Clasificadores basados en Análisis Discriminante Lineal

LDA, o, del inglés, *Linear Discriminative Analysis*, es una técnica usada comúnmente en clasificación y en problemas de reducción de dimensionalidad de conjuntos de datos. Se basa en la idea de maximizar la relación de varianza entre clases frente a la varianza dentro de la misma clase para un set de datos garantizando la máxima separabilidad de las clases. La principal diferencia entre LDA y PCA es que PCA cambia la forma y localización del set de datos original cuando realiza la transformación a diferentes espacios; mientras que LDA no cambia la localización, si no que trata de ofrecer una mejor separación de las clases dibujando una región de decisión entre ellas 2.10.

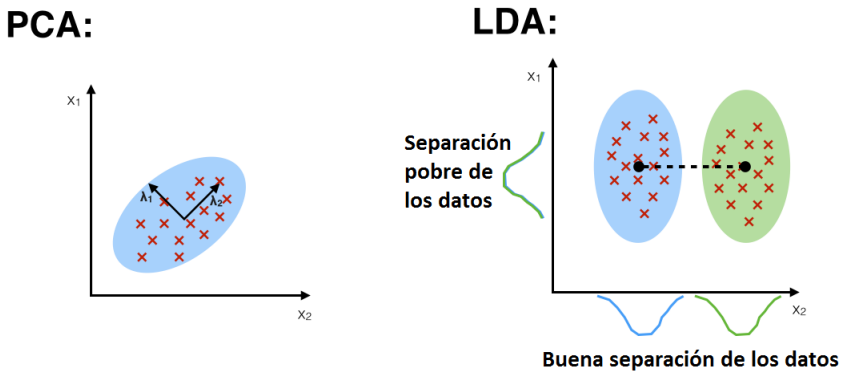

Figura 2.10: Diferencia entre PCA y LDA, mientras PCA busca las direcciones de máxima varianza de la clase, LDA busca la mejor separación entre clases.

Los datos de entrada pueden ser transformados, y por tanto, los vectores pueden ser clasificados en el espacio transformado mediante dos aproximaciones diferentes:

- **Transformación dependiente de la clase:** este tipo de aproximación trata de maximizar la relación de la varianza entre clases frente a la varianza dentro de las propias clases. El objetivo de esta maximización es obtener una separación adecuada de las clases, utilizando dos criterios de optimización para transformar los conjuntos de datos de forma independiente.
- **Transformación independiente de la clase:** en este caso se intenta maximizar la relación entre la varianza global y la varianza dentro de las clases. Sólo se utiliza un criterio de optimización para transformar los sets de datos, y por tanto, todos los datos independientemente de la clase a la que pertenecen.

Existen varias implementaciones de LDA, entre ellas, se encuentra Fisher-LDA [15]. Para su explicación se va a considerar la versión más simple del problema. En este caso se trata de encontrar el vector w de proyección, que proyecta los datos a un espacio uni-dimensional de manera que se pueda obtener la mayor separación entre clases.

Sabiendo que: $x_1..x_n$ son los patrones d-dimensionales, etiquetados en c clases, que cada clase cuenta con N_c patrones, se busca w para obtener y_i proyecciones unidimensionales de los patrones, siendo $y_i = w^T x_i$.

Mediante el método Fisher, se trata de maximizar la función:

$$J(w) = \frac{w^T S_B w}{w^T S_W w} \tag{2.35}$$

Donde S_B es la matriz de dispersión entre clases y S_W la matriz de dispersión dentro de la clase, es decir:

$$S_B = \sum_c N_c(\mu_c - \mu)(\mu_c - \mu)^T \tag{2.36}$$

$$S_W = \sum_c \sum_{i \in c} (x_i - \mu_c)(x_i - \mu_c)^T \tag{2.37}$$

Siendo μ_c la media de cada clase, μ la media de todos los datos y N_c la cantidad de patrones de la clase c.

Fisher-LDA busca el vector de w de proyección que maximiza el cociente entre estas matrices. Operando, se puede observar que el vector w debe cumplir que:

$$S_B w = \lambda S_W w \tag{2.38}$$

Si S_W es no singular, podemos resolver el clásico problema de valores propios para la matriz $S_W^{-1} S_B$ como:

$$S_W^{-1} S_B w = \lambda w \tag{2.39}$$

Y sustituyendo en 2.35 se obtiene:

$$J(w) = \frac{w^T S_B w}{w^T S_W w} = \lambda_k \frac{w_k^T S_B w_k}{w_k^T S_W w_k} = \lambda_k \; conk = 1..d \tag{2.40}$$

Siendo w_k el vector propio k de valor propio λ_k.

En consecuencia, para maximizar la solución debemos considerar el vector propio con mayor valor propio asociado. Este caso se ha desarrollado para la proyección de datos sobre un espacio uni-dimensional. Se puece ver sin mayores problemas que para el caso de querer proyectar sobre un espacio de dimensiones mayores m, se debe resolver el mismo problema y elegir los m vectores propios con valores propios asociados más grandes [6].

2.4.2 Clasificadores basados en Aprendizaje No Supervisado

El aprendizaje no supervisado es un método de aprendizaje automático donde un modelo es ajustado a ciertas observaciones. A diferencia del aprendizaje supervisado, en este caso no existe un conocimiento a priori, sino que se trata el conjunto de datos de entrada al sistema como un conjunto de variables aleatorias, con el fin de construir un modelo de densidad. El aprendizaje no supervisado puede ser utilizado para producir probabilidades condicionales para cualquiera de las variables aleatorias dadas. También es útil para la compresión de datos: fundamentalmente, todos los algoritmos de compresión dependen tanto explícita como implícitamente de una distribución de probabilidad sobre un conjunto de entrada. Otra forma de aprendizaje no supervisado es el clustering, o agrupación en inglés, el cual a veces no es probabilístico.

2.4.2.1 Redes neuronales no supervisadas

Como ya se ha comentado en 2.4.1.2, las redes neuronales pueden basar su aprendizaje en los métodos no supervisados. Estas redes no requieren influencia externa para modificar los pesos de las conexiones entre neuronas. La red tampoco recibe ninguna información por parte del entorno que le indique si la salida es correcta o no, por lo que existen varias posibilidades en cuanto a la interpretación de la salida en estas redes. En algunos casos, la salida representa el grado de similitud entre la información que se le está dando en la entrada y las informaciones que se le han mostrado en el pasado. En otros casos podría realizar una codificación de los datos de entrada, generando a la salida una versión codificada de esos datos, manteniendo la información relevante.

En general en este tipo de aprendizaje se consideran dos tipos:

- **Aprendizaje Hebbiano:** consiste básicamente en el ajuste de los pesos de las conexiones de acuerdo con la correlación. Es decir, si las dos unidades son activas, se produce un refuerzo de la conexión, mientras que si una es activa y la otra pasiva, esa conexión se debilita.
- **Aprendizaje competitivo:** las neuronas compiten unas con otras con el fin de llevar a cabo cierta tarea. Con este tipo de aprendizaje se pretende que cuando se presente a la red cierta información de entrada, sólo una de las neuronas de la salida se active. Por tanto las neuronas compiten por activarse, quedando finalmente una, o una por grupo, que alcanza su valor de respuesta máximo.

Los principales tipos de redes neuronales que basan su aprendizaje en el no supervisado son las siguientes.

Redes competitivas

Como su nombre indica, utilizan el aprendizaje competitivo para la representación de patrones. Las neuronas compiten para ver cuál representa mejor el patrón y la vencedora se lleva todo el aprendizaje de ese patrón. El objetivo de este tipo de redes es que se formen grupos de patrones, categorías, que son representados por cada neurona. Las redes competitivas son usualmente bicapas. La función de la primera capa es hacer de sensor. La segunda capa tiene tantas neuronas como categorías deseemos. Sin embargo, algunas redes competitivas como las de la familia ART, crean neuronas dinámicamente para ajustar el número de categorías de forma automática. La conexión de neuronas es del tipo todas con todas.

Redes competitivas ART

ART son las siglas en inglés de Teoría de la Resonancia Adaptativa. Esta teoría se formula en respuesta al dilema de la Plasticidad del aprendizaje frente a la Estabilidad. La plasticidad permite a una red neuronal aprender nuevos patrones, mientras que la estabilidad permite a la red retener los patrones ya aprendidos. Conseguir que una red neuronal resuelva uno de los dos problemas es relativamente sencillo, pero el reto está en conseguir un modelo que de respuesta a ambos. El modelo ART soluciona este dilema mediante un mecanismo de realimentación entre las neuronas competitivas de la capa de salida. Cuando a la red se le presenta un patrón de entrada, éste se hace resonar con los prototipos de las categorías conocidas por la red. Si este patrón entra en resonancia con alguna clase, entonces queda asociado a esa clase. Sin embargo, si no entra en resonancia, lo que ocurra a continuación dependerá del tipo de red. Si la capa de salida es estática, la red entra en saturación ya que no puede crear una nueva clase en la que ubicar el patrón, ni ubicarlo en niguna de las clases conocidas. Sin embargo, si la capa de salida es dinámica, se creará una nueva clase, sin afectar a las demás.

Redes de Kohonen

Las redes de Kohonen, también conocidas como mapas auto-organizados, son redes utilizadas para producir una representación discreta del espacio de las muestras de entrada, llamado mapa. Son diferentes a otras redes neuronales en el sentido en que usan una función de vecindad para preservar las propiedades topológicas del espacio de entrada 2.13. Estos mapas son útiles para visualizar datos de alta dimensión en vistas de baja dimensión, es decir, los mapas auto-organizados describen un mapeo de un espacio de mayor dimensión a uno de menor dimensión. Asociado a cada neurona hay un vector de pesos, de la misma dimensión de los vectores de entrada, y una posición en el mapa. Para ubicar un vector de

entrada en el mapa, se busca la neurona con el vector de pesos más cercano al vector de datos de entrada. Típicamente, se disponen las neuronas en un espacio regular de dos dimensiones, en una rejilla hexagonal o rectangular.

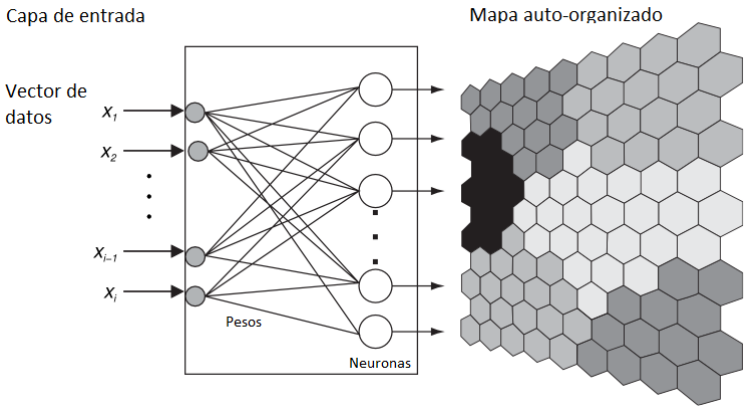

Figura 2.11: Representación gráfica de una red de Kohonen y su salida en forma de mapa de características.

2.4.2.2 Clustering

El análisis por agrupación o clustering es la técnica que agrupa conjuntos de objetos de tal manera que los que pertenecen al mismo grupo, o cluster, son más parecidos (de una u otra forma) entre sí que a los objetos de los demás grupos. El clustering es una de las principales técnicas utilizadas en el análisis estadístico de datos, utilizada en campos como el aprendizaje máquina, el reconocimiento de patrones, análisis de imágenes, compresión de datos, etc.

El clustering no se trata de un único algoritmo, sino que encontramos diferentes formas de abordar este problema. Comúnmente, los clusters se definen de distintas maneras: grupos con pequeñas distancias de separación entre los datos, áreas con mayor densidad, intervalos, o incluso distribuciones estadísticas particulares de los datos. Por todo ello, el clustering puede ser formulado como un problema de optimización multiobjetivo. El uso apropiado de uno de los algoritmos, así como la buena elección de los parámetros que lo definen (funciones distancia utilizadas, umbral de densidad, cantidad de grupos o clusters...) dependerán tanto del set de datos que tratemos, como de la utilidad que le vayamos a dar a los resultados. No se trata de un proceso automático si no de un proceso iterativo. Mediante la modificación de los parámetros de los modelos y la forma de procesar los datos, se buscan los resultados deseados a partir de un proceso de prueba y error.

El análisis mediante cluster tiene su origen en el campo de la antropología, originado por Driver y Kroeber en 1932, e introducido en psicología por Zubin en 1938 y por Robert Tryon en 1939 [16] [17]. En 1943 Cattell comenzó a utilizar esta técnica, también en el ámbito de la psicología, para su teoría de clasificación de la personalidad [18].

Es difícil definir con precisión el término cluster, una razón más por la que encontramos tantos algoritmos para el clustering.[19] Sin embargo, el denominador común es sencillo: un grupo de datos. Por tanto, la clave para entender esta técnica de agrupación de datos es entender los modelos de cluster que podemos encontrar, y que a su vez definen los distintos algoritmos a nuestra disposición. Los modelos de cluster más comunes son:

- **Modelos basados en distancias:** por ejemplo, con la agrupación jerárquica, con el fin de decidir qué grupos se deben combinar o dividir, se requiere una medida de disimilitud entre los conjuntos

de datos. Esto se logra mediante el uso de una medida de la distancia entre pares de datos, y un criterio de vinculación que especifica la similitud de los conjuntos como una función de las distancias por pares de datos dentro de cada conjunto.

- **Modelos basados en centroides:** mediante el algoritmo *k-means* se busca agrupar las observaciones en k grupos. Cada observación pertenecerá al grupo con valor medio más cercano. En este algoritmo se utiliza el centro del grupo para modelar los datos, y además, tiende a encontrar grupos de extensión espacial comparable.

- **Modelos basados en distribuciones de probabilidad:** los clusters se organizan siguiendo distintas distribuciones de probabilidad tales como la distribución multivariante normal usada en el algoritmo EM (Esperanza-maximización).

- **Modelos basados en densidad:** por ejemplo, los algoritmos DBSCAN [20] y OPTICS [21], dado un conjunto de datos en el espacio, definen los clusters por puntos muy juntos entre sí, regiones de alta densidad, y marca como valores atípicos aquellos que se encuentran muy alejados de cualquier dato, en lo que se llaman regiones de baja densidad.

También es importante remarcar que el clustering, entendiéndolo como la agrupación de los clusters definidos por el algoritmo, suele diferenciarse en dos: *hard-clustering*, en el que cada objeto del espacio pertenece o no a un cluster; y el *soft-clustering*, en el que cada dato puede pertenecer a más de un cluster, definiendo para cada dato el grado de pertenencia a dichos grupos.

Los algoritmos de clustering se categorizan por el modelo de cluster que utilizan, tal y como se ha explicado anteriormente. Cabe destacar que no existen algoritmos objetivamente correctos para la agrupación de datos, si no que el más apropiado para resolver un problema en concreto se debe elegir mediante experimentación, y dependerá del set de datos, o de ciertas propiedades matemáticas aplicables al problema que nos indiquen si un modelo es el más apropiado. Así mismo, un algoritmo diseñado para cierto modelo no puede ser usado en sets de datos que contienen modelos distintos, por ejemplo, el algoritmo *k-means* no puede encontrar clusters no-convexos [19].

A continuación se procede a explicar los algoritmos más comúnmente utilizados, ya que podemos encontrar más de 100 algoritmos publicados y constatados para el clustering, y no todos ellos proveen modelos concretos para los clusters utilizados, lo que hace aún más difícil su categorización.

Clustering jerárquico

Estos algoritmos se basan en la idea de que cada objeto está más relacionado o es más similar a objetos cercanos a él que a los más alejados. Por tanto, el algoritmo conecta los datos para formar grupos o cluters basados en la distancia entre ellos, por lo que el cluster viene definido por la máxima distancia necesaria para conectar los datos. Cambiando esta distancia máxima se generan distintos clusters, que pueden ser representados mediante diagramas en árbol 2.12, lo que explica el nombre del algoritmo: el algoritmo no ofrece una única división de los datos, si no una extensa jerarquía de clusters que se mezclan unos con otros a ciertas distancias. En el diagrama, el eje Y representa las distancias a las que los clusters se unen, mientras que en el eje X se posicionan los objetos de tal manera que los clusters no se mezclan.

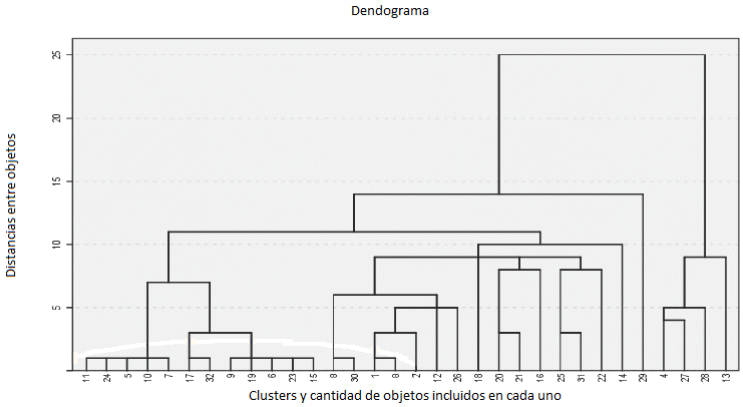

Figura 2.12: Representación del clustering jerárquico como diagrama de árbol.

El clustering jerárquico realmente define una familia de algoritmos que difieren unos de otros en la forma de calcular las distancias que definen a los clusters 2.13. A parte de el uso de funciones distancia, el usuario debe decidir el criterio de unión entre datos a utilizar. Los más comunes son: *single linkage clustering*, que utiliza la distancia mínima posible para las agrupaciones; *complete linkage clustering*, que al contrario que el anterior, utiliza la distancia máxima; o UPGMA (del inglés Unweighted Pair Group Method with Arithmetic mean) que utiliza la distancia media entre pares de datos para generar clusters.

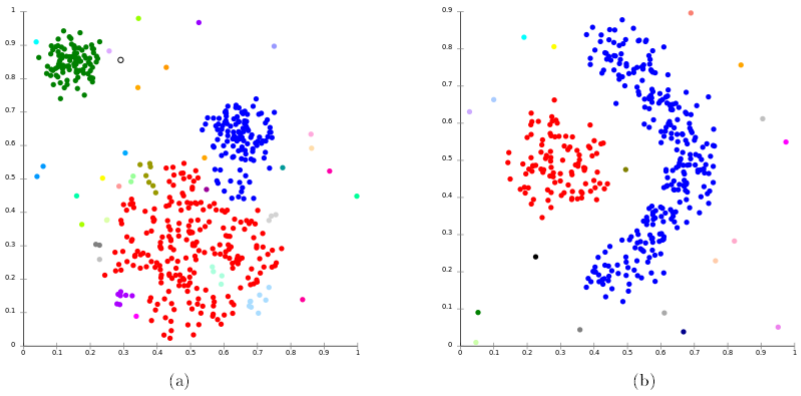

Figura 2.13: Representación del clustering jerárquico: (a) Sobre una distribución Gaussiana. (b) En este gráfico se observa que este modelo no hace distinción sobre el ruido, si no que lo incluye en clusters con un único elemento.

K-means clustering

Este algoritmo se basa en representar los clusters mediante un vector, que no tiene por qué formar parte del set de datos. Cuando el número de clusters se ajusta a k, el algoritmo da una definición formal como problema de optimización: encontrar los k centros que definen los clusters e ir asignando cada dato al cluster con centro más cercano, de tal forma que las distancias cuadráticas a éstos sean minimizadas.

Este problema de optimización resulta ser bastante complejo, por lo que se enfoca a encontrar soluciones aproximadas. El algoritmo *k-means* encuentra un máximo o mínimo local, y normalmente se ejecuta varias veces con distintas inicializaciones. Las distintas variantes del algoritmo se centran en encontrar

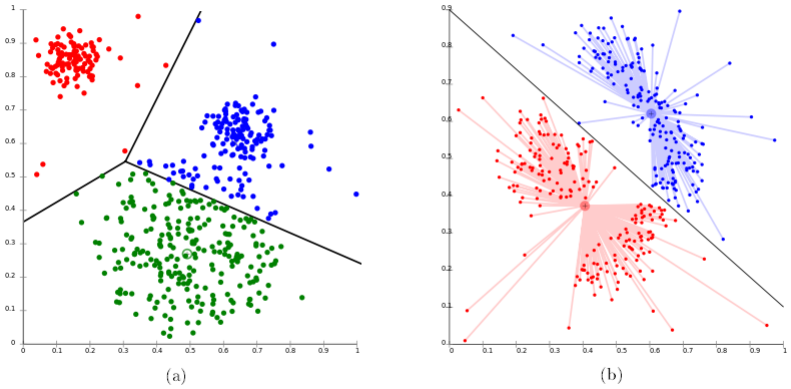

Figura 2.14: Representación del *K-means* clustering: (a) Ejemplo de la separación que hace este algoritmo (b) El algoritmo k-medios no es capaz de representar clusters basados en densidad.

la mejor de las distintas ejecuciones, restringir los centroides a datos pertenecientes al set o seleccionar los centroides de una forma menos aleatoria. La mayoría de este tipo de algoritmos requiere especificar el número de clusters k previamente, lo que resulta ser un gran inconveniente según la aplicación.

Clustering basado en distribuciones

Este algoritmo es el que más relacionado está con la estadística. Los clusters se definen mediante los objetos que mejor se ajustan a la misma distribución. La principal ventaja de este modelo es que se parece mucho a la forma en la que se generan los conjuntos de datos artificiales: mediante el muestreo al azar de objetos de una distribución.

El fundamento teórico de este algoritmo es irrefutable, sin embargo, presenta un gran inconveniente, el sobreajuste. Este problema se puede solucionar imponiendo ciertas restricciones que incrementan la complejidad del problema. El compromiso entre la necesidad de un modelo que defina y agrupe los datos suficientemente bien, y la complejidad que éste adquiere, resulta ser el principal inconveniente de este algoritmo. Uno de los métodos más utilizados se basa en las distribuciones gaussianas. En ellos, el set de datos se modela con un número concreto de distribuciones para evitar el sobreajuste, inicializándolas de forma aleatoria y ajustando sus parámetros de forma iterativa para encajar mejor en el set de datos.

Estos algoritmos producen modelos complejos de los clusters, incluso llegando a introducir correlaciones y dependencias entre atributos. Además, para muchos sets de datos reales, es posible que no podamos encontrar un modelo matemático que defina las distribuciones.

Clustering basado en densidad

Esta forma de clustering o agrupación define los clusters mediante áreas en las que la densidad de datos en el espacio es mayor. Los datos que se encuentran en áreas de menor densidad, generalmente se consideran ruidos o puntos de frontera, y son necesarios para poder separar los clusters.

El algoritmo basado en densidad más utilizado es DBSCAN [20]. De forma parecida al clustering jerárquico, se basa en conectar los puntos más cercanos dentro de cierta distancia mínima. La diferencia reside en que sólo enlaza los puntos que satisfacen además un criterio de densidad. Es decir, además de estar a cierta distancia, debe haber un número mínimo de objetos alrededor dentro de esa distancia

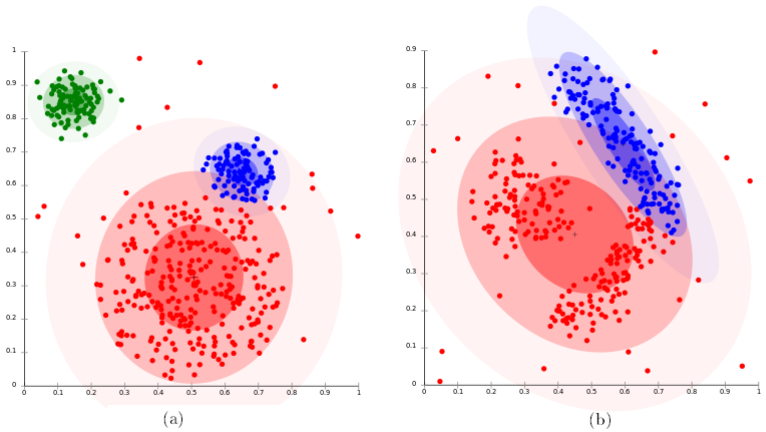

Figura 2.15: Representación del clustering basado en distribuciones: (a) Sobre una distribución Gaussiana el algoritmo funciona correctamente ya que se utilizan también modelos Gaussianos para los clusters. (b) Este algoritmo no permite representar clusters basados en densidad.

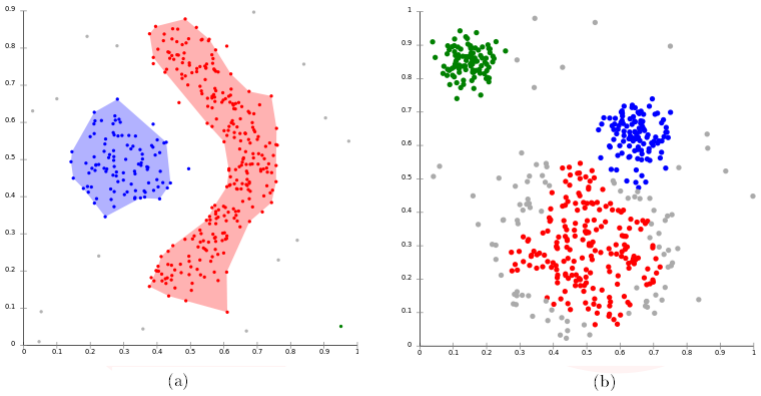

Figura 2.16: Representación del clustering basado en densidad: (a) El algoritmo DBSCAN es adecuado para la misma distribución de datos que en las figuras 2.14 b) y 2.15 b) que no eran capaces de separar correctamente. (b) Cuando se asumen clusters con la misma densidad, DBSCAN puede tener problemas separando clusters cercanos.

mínima o radio mínimo (definido por el usuario). A diferencia de otros algoritmos, DBSCAN produce clusters bien definidos, sin importar el tamaño o forma de cada uno 2.16. Otra de las ventajas de este método es que su complejidad es relativamente baja, y que proporciona básicamente los mismos resultados independientemente de las veces que se ejecute. OPTICS [21] generaliza DBSCAN para eliminar la necesidad de establecer el valor del radio que define los clusters. En cambio, produce un resultado jerárquico basado en las agrupaciones y las distancias utilizadas.

2.4.3 Clasificadores basados en Aprendizaje Semi-Supervisado

El aprendizaje semi-supervisado es un conjunto de técnicas relacionadas con el aprendizaje supervisado que además, también utilizan datos no etiquetados para el entrenamiento. Generalmente utilizan una pequeña cantidad de datos etiquetados y un gran set de datos sin etiquetar.[22] Muchos estudios sobre el aprendizaje máquina han revelado que la utilización de esta mezcla entre datos etiquetados y sin etiquetar provoca mejoras sustanciales en el aprendizaje y su exactitud. Conseguir datos etiquetados para un problema concreto de aprendizaje requiere normalmente agentes humanos capacitados para clasificar

el set de datos. Este proceso puede resultar muy costoso, y para determinada cantidad de datos, inviable. Sin embargo, la adquisición de datos sin clasificación previa no resulta ni difícil ni costosa, por lo que el aprendizaje semi-supervisado adquiere un gran valor práctico en estas situaciones.

Otra forma de enfocar esta supervisión parcial es mediante la utilización de restricciones. Esta forma de aprendizaje semi-supervisado se conoce como aprendizaje no supervisado guiado por restricciones. El enfoque comentado en el apartado anterior se adapta mejor a las aplicaciones en las que el objetivo es el de predecir la clase o el valor de cierto dato de entrada. Sin embargo, no es de tan fácil aplicación cuando el número y tipo de clases no se conoce previamente y debe ser inferido a partir del set de datos. En estos casos, la utilización de restricciones de aprendizaje semi-supervisado es el modelo que mejor se adapta.

Esta forma de aprendizaje, a su vez, presenta dos formas de aprendizaje: transductivo e inductivo. La idea del aprendizaje transductivo se basa en realizar predicciones sólo para los datos de test. El aprendizaje inductivo, sin embargo, trata de encontrar una función de predicción válida para todo el espacio de datos. La mayoría de métodos de aprendizaje semi-supervisado se basan en la transducción que, de hecho, es la forma más natural para la inferencia basada en representaciones gráficas de datos.

Para que la mejora que tiene este tipo de aprendizaje sobre el aprendizaje supervisado sea relevante, hay un requisito importante que debe cumplirse: que la distribución de los ejemplos, que los datos no etiquetados nos ayudará a resolver, sea importante para el problema de clasificación. Explicado desde un punto de vista más matemático, podemos decir que el conocimiento de $p(x)$ que se obtiene a partir de los datos no etiquetados debe contener información útil para poder deducir $p(y|x)$. Por tanto, para que este sistema de aprendizaje realmente funcione y mejore otros modelos, los algoritmos deben basarse en una de las siguientes suposiciones:

- **Suposición de suavidad:** si dos puntos x_1 y x_2 de una región de gran densidad de datos están cerca, sus correspondientes salidas y_1 , y_2 también lo estarán. Mientras que en el caso supervisado se asume que las salidas varian suavemente con la distancia, en este caso además se tiene en cuenta la densidad de datos de entrada en esa región.
- **Suposición de agrupación:** si dos puntos pertenecen a la misma agrupación (cluster), es probable que pertenezcan a la misma clase. Esta suposición no implica que cada clase forma un único cluster compacto; simplemente significa que, normalmente, no observamos objetos de distinta clase dentro de un mismo cluster. La suposición de agrupación se puede ver como un caso especial de la anterior, considerando que los clusters normalmente se definen como conjuntos de datos o puntos que pueden ser conetados por curvas que sólo atraviesan zonas con gran densidad.
- **Suposición de variedad:** los datos de alta dimensionalidad se apoyan en un espacio de dimensiones mucho menores. Si los datos se apoyan sobre un espacio dimensional menor, el algoritmo puede operar en un espacio de dimensiones equivalentes, evitando la elevada complejidad que puede llegar a tener el trabajar con datos de alta dimensionalidad.

Aunque existen distindos modelos que no se derivan explícitamente de las suposiciones comentadas, la mayoría de los algoritmos se corresponden o implementan uno o varios de ellos. A continuación se hace un breve comentario de los modelos más comunes, que sí se corresponden de forma más o menos aproximada a las suposiciones del aprendizaje semi-supervisado.

2.4.3.1 Modelos generativos

Los modelos generativos tienen como objetivo encontrar una estimación de la distribución de los datos y su pertenencia a una clase $p(x|y)$. Entonces, la probabilidad de que cierto punto x sea asociado a la

etiqueta y, $p(y|x)$ es proporcional, por la regla de Bayes, a $p(x|y)p(y)$. El aprendizaje semi-supervisado mediante modelos generativos se puede afrontar desde dos puntos de vista: como una extensión del aprendizaje supervisado, en el que se realiza la clasificación con información adicional sobre $p(x)$; o como una extensión del aprendizaje no supervisado, en el que además del clustering, tenemos información sobre las etiquetas de cada clase.

Los modelos generativos asumen que las distribuciones adquieren una definición particular, parametrizada por un vector θp(x|y,θ). Si esta suposición no es correcta, los datos sin etiquetar pueden hacer empeorar la precisión del sistema frente a uno que se hubiera implementado utilizando únicamente datos etiquetados. Los datos no etiquetados se disponen según una mezcla de distribuciones. Para poder resolver esa mezcla de distribuciones, deben ser identificables, es decir, la suma de las distintas distribuciones debe ajustarse a los distintos parámetros. Por ejemplo, la combinación de distribuciones Gaussianas es identificable, y resulta ser típica en los modelos generativos.

2.4.3.2 Modelos basados en separación de regiones de baja densidad

La suposición de agrupación puede ser formulada de forma equivalente de la siguiente manera: la frontera de decisión entre clases debe localizarse en regiones de baja densidad. Los modelos basados en esta reformulación intentan implementarla directamente forzando que los límites de decisión se encuentren alejados de los datos sin etiquetar.

La aproximación más común para conseguirlo es utilizar algoritmos de máximo margen como las máquinas de soporte vectorial (2.4.1.4). Este método de maximización del margen tanto para datos etiquetados como no etiquetados se conoce como TSVM, SVM transductiva. En resumen, partiendo de la solución dada por la SVM entrenada sólo para lo datos etiquetados, los no etiquetados se le pasan a la SVM para que los clasifique, para posteriormente reentrenar la SVM con todos los datos. Otras alternativas a TSVM incluyen una clase "nula"que ocupa el espacio entre dos clases corrientes, ofreciendo la ventaja de que permite soluciones probabilísticas. Esta ventaja también la ofrece un modelo conocido como el de minimización de entropía. Éste fuerza las probabilidades condicionales de las clases $P(y|x)$ a aproximarse a 1 ó 0 para datos etiquetados y sin etiquetar. Como consecuencia de la suposición de suavidad, esta probabilidad tenderá a 1 ó 0 en las regiones de alta densidad, mientras que tomará valores intermedios en las fronteras entre clases.

2.4.3.3 Modelos basados en grafos

El denominador común de estos modelos es que los datos están representados por nodos de un grafos, donde los ejes están definidos como distancias a los nodos por pares de nodos. Si la distancia entre dos puntos se calcula minimizando la trayectoria total sobre todas las trayectorias que conectan los dos puntos, podemos hablar de una aproximación de la distancia geodésica entre dos puntos respecto al espacio de datos reducido, es decir, la suposición de variedad.

Normalmente las predicciones consisten en etiquetar los datos no etiquetados, por lo que se dice que este tipo de algoritmos son puramente transductivos. Devuelven únicamente el valor que la función de decisión obtiene para estos datos, pero no la función como tal, aunque trabajos recientes se enfocan en conseguir que los modelos de gráfico consigan soluciones inductivas. La propagación de información en los gráficos también puede servir para mejorar una clasificación ya dada (probablemente supervisada) teniendo en cuenta los datos sin etiquetar.

Capítulo 3

Desarrollo

A fuerza de construir bien, se llega a buen arquitecto

Aristóteles

3.1 Introducción

El objetivo del proyecto consiste en la ampliación del sistema de detección de personas implementado en [3] para resolver los problemas que se dan cuando ocurren oclusiones en la imagen, es decir, cuando las personas que tratamos de localizar en las imágenes se encuentran parcialmente ocultas por otro objeto. Para ello, se ha tratado de implementar un sistema de detección que se centra en la mitad superior de las personas, ya que la combinación de detecciones de persona completa y media persona resulta ser el modelo más efectivo y rápido para lidiar con el problema de oclusión [23] [24].

Este proceso se desarrolla en tres fases, tal y como se puede observar en el diagrama de bloques de la figura 3.1: extracción de descriptores HOG, entrenamiento de la SVM e implementación del algoritmo de detección.

En la primera fase se eligió un set de imágenes de entrenamiento válido para nuestra aplicación y se escribió el algoritmo que permite extraer los descriptores de las mismas. Además del set de entrenamiento, fue necesario un set de imágenes de test que permitiera comprobar el funcionamiento del sistema completo y observar si los resultados que se obtenían eran satisfactorios.

En la segunda se utilizaron estos descriptores para entrenar una SVM que se adaptara a los requisitos de la aplicación, y por último, se probó la capacidad del sistema mediante un algoritmo de detección que localizara la parte superior de personas en el set de imágenes de test.

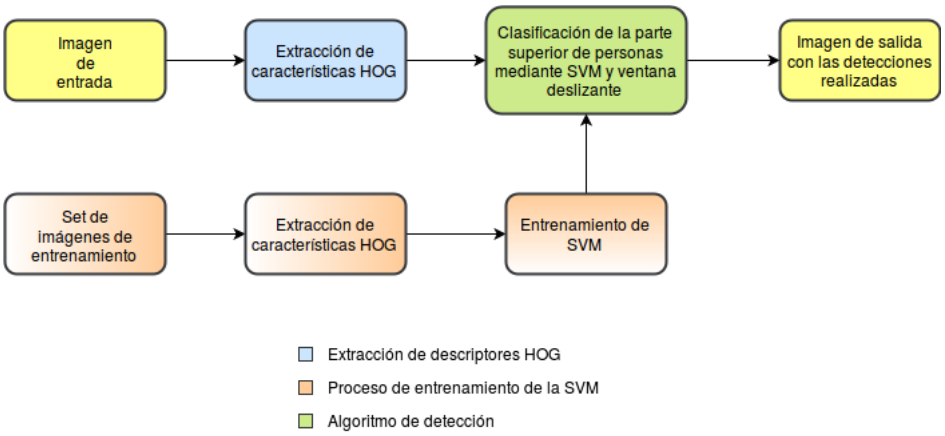

Figura 3.1: Esquema simplificado de las fases desarrolladas en el proyecto.

En los siguientes apartados se describe el trabajo desarrollado para la implementación de los distintos algoritmos para cada fase, así como la justificación de las distintas decisiones de diseño, selección de imágenes, configuración de parámetros, etc. Además, por cada apartado se detallan las principales conclusiones extraídas del mismo.

3.2 Extracción de Descriptores HOG

Como se ha comentado en la introducción, en esta fase se procedió a la extracción de descriptores HOG 2.3 de un set de imágenes para su posterior uso en el entrenamiento de una SVM.

Lo primero que se hizo fue obtener un set de imágenes de la base de datos de INRIA en la que aparecen personas centradas en la imagen. El tamaño de estas imágenes era de 96x160 píxeles, por lo que era necesario redimensionarlas debido a los requisitos de HOG que más adelante serán comentados. Además, fue necesario recortar las imágenes para que en ellas apareciera la mitad superior de la persona, y no la persona completa. Para ello, se realizó un script (*resizeimages.cpp*) que tomaba las imágenes del directorio deseado, ajustaba su tamaño a 64x128, definía una ROI de 64x64 píxeles que encuadraba la mitad de la persona, la recortaba y la guardaba en un nuevo directorio. Para realizarlo, también se utilizó el programa KRename [25] que permite renombrar todos los archivos de un directorio con el nombre e índice deseados, y manteniendo un orden, lo que facilitaba definir el bucle principal que tomaba las imágenes del directorio para modificarlas. En total se obtuvieron 2416 imágenes positivas, es decir, de la clase media persona. Mediante el mismo proceso (*resize_negimages.cpp*), sin la necesidad de recortar las imágenes, sino simplemente redimensionarlas, se obtuvo el set de imágenes negativas, compuesto por 1218 imágenes en las que aparece cualquier cosa que no sea la mitad superior de una persona; ya sean edificios, calles vacías, árboles o incluso bicicletas (3.2).

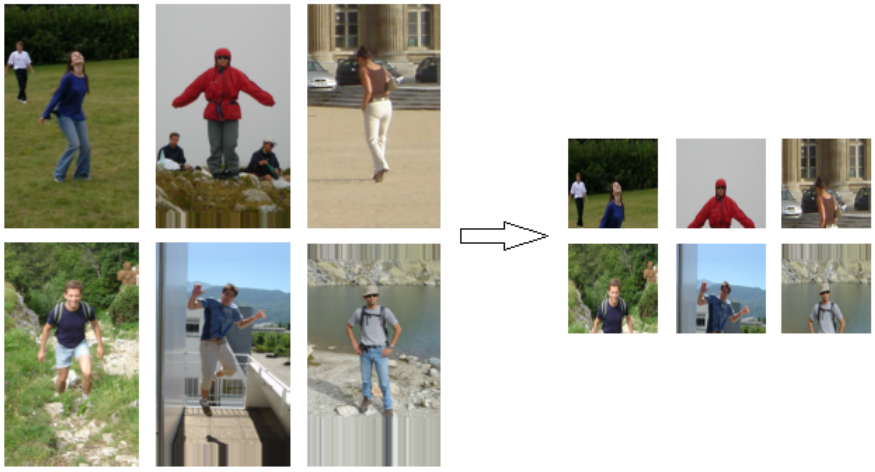

Figura 3.2: Comparación entre algunas imágenes de la base de datos de INRIA e imágenes del set utilizado en la aplicación.

Una vez obtenido el set de imágenes de entrenamiento, se procedió a la extracción de los descriptores HOG de cada imagen necesarios para entrenar posteriormente la SVM. Para ello se escribió *pos_recimgs_descript.cpp*, un programa que definía un objeto de la clase HOG, accedía al directorio en el que se encontraban las muestras positivas, y mediante el método *hog.compute*, extraía de cada imagen su descriptor. Cada descriptor consiste en un vector con un cierto número de componentes que definen las características de apariencia y forma de la imagen. Estos vectores eran almacenados en un archivo de texto para su posterior utilización en las siguientes fases del sistema. De la misma manera, mediante *neg_imgs_descript.cpp* se realizaba el mismo proceso para obtener los descriptores del set de imágenes negativas y guardarlos en su propio archivo de texto.

3.2.1 HOG y hog.compute en detalle

Para entender tanto la elección de 64x64 como tamaño de imagen, como las dimensiones del descriptor de cada imagen entregado por el método *hog.compute*, es necesario explicar con detenimiento la clase HOG y sus atributos, así como el propio método utilizado.

Fragmento de código 3.1: Definición de un objeto de HOG.

```
HOGDescriptor hog (Size(64,64), Size(16,16), Size(8,8), Size(8,8), 9, -1,
    HOGDescriptor::L2Hys, 0.2, false, HOGDescriptor::DEFAULT_NLEVELS, false);
```

La clase HOG implementa el descriptor ideado por Navneet Dalal [5], ya comentado en 2.3. Sus parámetros definen los tamaños de ventana, celdas y bloques, así como la forma de procesar algunos de los métodos de la clase [26]. A continuación se describen por orden y en detalle los parámetros mostrados en el fragmento de código 3.1:

- **Tamaño de ventana = win_Size Size(64x64):** se mide en píxeles y define el tamaño de la ventana a ser analizada. Debe coincidir con el tamaño de la imagen que se le pasa al algoritmo, y además, concordar con el tamaño del bloque y con su desplazamiento. De tal forma, las imágenes del set de entrenamiento tienen el tamaño de 64x64 para hacerlo coincidir con el tamaño de ventana de HOG.

- **Tamaño de bloque = block_Size Size(16x16):** también en píxeles, define el tamaño del bloque utilizado en la fase de normalización del algoritmo. Debe alinearse con el tamaño de celda, y el único tamaño soportado por la librería de *OpenCV* es el de 16x16, por ahora.

- **Desplazamiento del bloque = block_Stride Size(8x8):** definido en píxeles, fija el desplazamiento del bloque en cada iteración de la fase de normalización del algoritmo. Además, su tamaño debe ser múltiplo del tamaño de celda.

- **Tamaño de celda = cell_Size Size(8x8):** define el tamaño de la celda. Ésta es la agrupación de píxeles primaria en el algoritmo. A partir de ella se obtienen los vectores de gradientes, valores básicos y fundamentales del algoritmo. *OpenCV* sólo admite 8x8 por el momento.

- **Número de contenedores = nbins 9:** define la cantidad de grupos en los que se divide el histograma de gradientes de cada celda. 9 es el número por defecto de contenedores, además de ser el único que soporta *OpenCV* por el momento.

- **Parámetro de suavizado Gaussiano = win_sigma:** parámetro utilizado en el procesado de la imagen que realiza el algoritmo antes de empezar el análisis y la extracción de descriptores. Su valor por defecto definido en la clase es -1.

- **Constante de normalización del bloque = threshold_L2hys :** constante utilizada en el método de normalización, por defecto 0.2 y como única opción, la norma L2 con truncamiento [3].

- **Corrección gamma = gamma_correction:** variable booleana que indica si es necesaria y por tanto se realiza la corrección gamma en el preprocesado de la imagen con el objetivo de reducir la influencia de la iluminación. En nuestro caso su utilización no influía notablemente en el sistema por lo que se decidió desactivar esta parte del preprocesado de imágenes.

- **Número máximo de ventanas de detección = nlevels:** indica el número máximo de escalas que aplicará HOG a la ventana de detección (importante en el método *detectMultiScale*[3.4.1]). El valor por defecto establecido por *OpenCV* es 64.

Estos son los parámetros que definen la clase HOG y que influyen en los distintos métodos que implementa la clase. A continuación, se describen los parámetros del método *hog.compute*, utilizado para extraer los descriptores de las imágenes, así como su funcionamiento [26].

Fragmento de código 3.2: Método *hog.compute*.

```
hog.compute(img, pos_rec_descript, Size (8,8), Size (0,0),locations);
```

- **Imagen = img:** imagen de entrada que se le pasa a la función. Esta es la imagen que será analizada y de la que se extraerán los descriptores, y su tamaño debe coincidir con el tamaño de ventana definido en el objeto previamente.

- **Descriptores = descriptors:** vector donde se guardan todos los valores de los histogramas de gradientes de la imagen extraídos por el algoritmo, y que definen las características de apariencia y forma de la imagen.

- **Desplazamiento de ventana = win_Stride Size(8,8):** define el desplazamiento de la ventana en la que se calculan los gradientes píxel a píxel. Debe coincidir con el tamaño de la celda definido en el objeto para no perder información de la imagen.

- **Relleno = padding Size(0,0):** define la cantidad de píxeles que se añaden alrededor de la imagen. De esta forma, cuando la ventana pasa por los bordes, no coinciden exactamente los bordes de la ventana con los bordes de la imagen. De manera que la ventana incluye además este relleno exterior que no forma parte de la imagen. La utilidad de este parámetro reside en la capacidad que otorga al algoritmo de estudiar con mayor precisión los bordes de la imagen. En nuestro caso, los distintos valores del parámetro no mejoraban los resultados del sistema global por lo que se decidió dejarlo a cero.

- **Localizaciones = locations:** parámetro que almacena la región de la imagen en la que se extraen los descriptores.

Para calcular cada descriptor de cada imagen, *hog.compute* trabaja con las celdas de 8x8 píxeles. Dentro de cada celda, se calcula el vector gradiente de cada píxel, y se colocan los 64 vectores obtenidos en un histograma con 9 contenedores. El rango del histograma va de 0 a 180 grados, por lo que cada contenedor representa 20 grados [3.3]. La magnitud de cada vector es la que contribuye al histograma y se divide entre los dos contenedores adyacentes. Es decir, si por ejemplo tenemos un vector gradiente con un ángulo de 85 grados, un cuarto de su magnitud aportará peso al contenedor centrado en los 70 grados, y los otros tres cuartos de su magnitud, al de 90 grados. Con esto evitamos el problema de los gradientes que se encuentran justo entre dos contenedores, ya que si obtenemos un gradiente de magnitud importante en ese punto, pequeños cambios en su ángulo lo harían pasar de un contenedor a otro, pudiendo esto tener un fuerte impacto en el histograma.

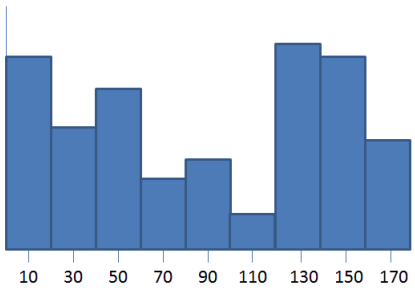

Figura 3.3: Ejemplo de un histograma de gradientes de 9 contenedores.

El siguiente paso del algoritmo consiste en la normalización. En lugar de normalizar cada histograma por separado, las celdas se agrupan en bloques y la normalización tiene en cuenta todos los histogramas del bloque. La figura 3.4 ilustra las dimensiones y distribución, tanto de las celdas como de los bloques, en una imagen del set. Para llevar a cabo dicha normalización, se concatenan los histogramas de las 4 celdas que forman el bloque en un vector de 36 componentes (4 histogramas con 9 contenedores cada uno, como se define en el fragmento de código 3.1). Una vez obtenido este vector, se divide por su magnitud para normalizarlo. El parámetro de desplazamiento del bloque explicado anteriormente en la definición del objeto provoca una superposición entre bloques, como se puede observar en la figura 3.4. El efecto de esta superposición es que cada celda aparece varias veces en el descriptor final, pero normalizada en base a un distinto grupo de celdas adyacentes. En concreto, las celdas de las esquinas aparecen una única vez, las de los bordes dos, y el resto de celdas interiores aparecen cuatro veces cada una. Mediante este proceso se consigue una mayor invarianza y robustez ante cambios en el contraste o en la iluminación.

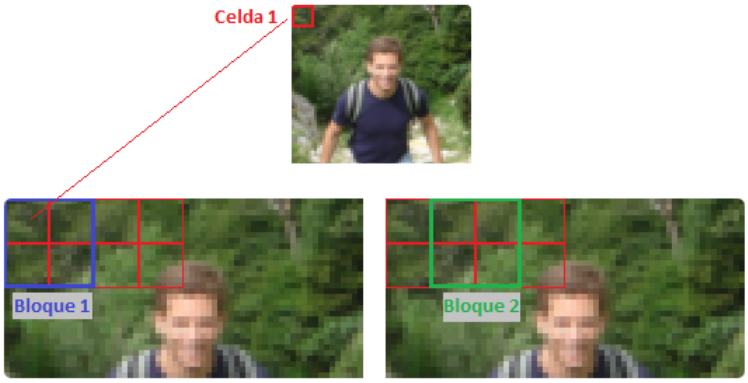

Figura 3.4: Visualización de las celdas, bloques y el solapamiento entre ellos que implementa *hog.compute*.

Una vez completada la extracción del descriptor de toda la imagen, tenemos un vector con un número concreto de componentes. La ventana de detección (que recordemos, debe coincidir con el tamaño de la imagen a analizar) de 64x64 píxeles queda dividida en 7 bloques de ancho por 7 de largo, para un total de 49 bloques. Cada bloque contiene 4 celdas con un histograma de 9 contenedores cada una. Por tanto, 7 bloques de alto x 7 bloques de ancho x 4 celdas por bloque x 9 contenedores por celda, hacen un descriptor de 1764 componentes.

3.2.2 Conclusiones

Recapitulando, al terminar esta fase se tenían dos archivos de texto con los descriptores de cada imagen positiva y negativa. Es decir, 2416 vectores de 1764 valores cada uno agrupados en un fichero, que definían el set de imágenes positivas; de igual modo que se obtenían 1218 vectores con el mismo número de valores para definir las imágenes negativas. Ésta es una cantidad considerable de datos y, al ser manejados en la siguiente fase, el orden establecido en su almacenamiento y posterior utilización cobraba una importancia mayor, ya que errores en su utilización podían afectar en gran medida al funcionamiento del sistema completo.

También cabe destacar la importancia que adquiere la definición correcta de los parámetros, tanto del objeto de la clase, como de los métodos utilizados. Ya que gran parte de los parámetros son parámetros por defecto, y *OpenCV* no está preparado para utilizar otros valores de esos parámetros; se hacía muy importante mantener la concordancia de los parámetros entre el objeto hog y el método *hog.compute* para conseguir resultados ya no sólo correctos, sino además, coherentes.

3.3 Entrenamiento de la SVM

Una vez obtenidos los descriptores del set de imágenes, tanto positivas como negativas, y almacenados en sus correspondientes archivos de texto, se procedió a la implementación de una SVM; un tipo de clasificador basado en el aprendizaje supervisado, explicado en 2.4.1.4. Para ello se empezó por probar con ejemplos de código extraídos de [27] con el objetivo de aprender las herramientas que *OpenCV* pone a nuestra disposición para tratar con máquinas de soporte vectorial.

Este ejemplo entrenaba una SVM para la detección y clasificación de tuercas, tornillos y arandelas. Para ello utilizaba el mismo procedimiento que se sigue en este proyecto: extracción de descriptores de

las imágenes representativas de cada clase, entrenamiento de una SVM con estos descriptores y posterior detección y clasificación en imágenes. A la hora de compilar este código, los problemas fueron apareciendo: el primero de todos fue un problema de versiones de la librería, el código no podía ser compilado si no se tenía la versión 3.0 de la librería. Estando instalada la versión 2.4 en el ordenador de trabajo, hubo que actualizar la librería. Una vez actualizada, el compilador detectaba numerosos errores, sobre todo en la definición de los parámetros de la SVM. Esto ocurrió porque el código estaba escrito con las funciones que *OpenCV* implementaba en su versión 2.4, pero que cambiaron por completo en la versión 3.0. En resumen, el código no podía compilarse sin la versión 3.0 de la librería, pero estaba escrito con las funciones de la versión 2.4. Entonces se empezó a trabajar en el estudio de ese código y su modificación para adaptarlo a la versión correspondiente. Sin embargo, las modificaciones que se hacían daban lugar a distintos errores, muchos sin una solución inmediata o fácil de encontrar. Ante esto, se decidió dejar de lado este código y empezar de nuevo con otro ejemplo de base.

El ejemplo escogido fue un tutorial extraído de la web de documentación de *OpenCV* [28]. Se trata de un ejemplo básico en el que se crea un set de datos formado por 4 puntos del plano 2D, tres de la misma clase, y el cuarto de una distinta. El ejemplo ilustra cómo se deben preparar los datos y sus etiquetas para el entrenamiento de una SVM, además de mostrar qué tipos de SVM se pueden utilizar y cómo definir sus parámetros. Tanto la compilación como la ejecución de este ejemplo no dieron problemas, y debido a su sencillez y utilidad, se decidió estudiar y modificar el código para adaptarlo a nuestra aplicación.

Una vez creada una SVM del tipo indicado, con el kernel (3.3.1)necesario para la aplicación, el siguiente paso en el procedimiento fue el del entrenamiento. Para ello, era imprescindible que el set de descriptores extraído anteriormente estuviera organizado de forma correcta en relación con las etiquetas que lo describían. En este caso, la dificultad residió en la creación de las matrices requeridas para el entrenamiento de la SVM. Como se explica más adelante, el método *train* de la clase SVM requiere de dos matrices y un criterio. Una de las matrices debe contener las muestras de entrenamiento, es decir, en cada una de las filas se debe encontrar un único vector de características que define una única imagen, ya sea positiva o negativa. En la otra matriz cada fila representa una etiqueta (1 para muestras positivas, o -1 para negativas en el caso particular de nuestra aplicación). El lugar que ocupa una muestra en la matriz de características debe corresponderse con el que ocupa su etiqueta en la matriz de etiquetas.

Para concretar lo descrito anteriormente se implementaron dos bucles. Uno de ellos se encargaba de recorrer los archivos de texto en los que se habían guardado los descriptores de las imágenes de entrenamiento, leerlos y guardarlos en la matriz de entrenamiento. Resultaba importante mantener el orden en el que se recorrían los archivos de texto para tener organizadas en la matriz primero las muestras positivas y posteriormente las negativas. Del mismo modo, el segundo bucle rellenaba una matriz con la etiqueta positiva (1) tantas veces como imágenes de media persona se habían utilizado en el set de entrenamiento. Una vez llegado a ese número, la matriz se completaba con tantas etiquetas negativas (-1) como imágenes de entrenamiento de la clase 'no persona' se habían utilizado. De esta forma las dos matrices quedaban creadas y se podía proceder al entrenamiento de la SVM.

3.3.1 SVM y svm.train en detalle

La elección de una SVM como clasificador para nuestra aplicación se basa en la sencillez y robustez que ofrecen, así como en la necesidad de la aplicación de un clasificador binario que ofrezca buenas respuestas ante el tipo de datos que ha de tratar.

La librería *OpenCV* implementa una clase SVM que permite modificar los distintos parámetros que definen una SVM, como su tipo o su kernel, de manera rápida y sencilla. A continuación se presentan en detalle las distintas funcionalidades de la clase [29]:

Fragmento de código 3.3: Definición de un objeto de SVM.

```
Ptr<SVM> svm = SVM::create();
svm->setType(SVM::C_SVC);
svm->setKernel(SVM::LINEAR);
svm->setTermCriteria(TermCriteria(TermCriteria::MAX_ITER, 1000, 1e-6));
```

- **Tipo = svmType:** define las características de la SVM. Las distintas posibilidades son:
  - **C_SVC:** SVM tipo C. El margen de la SVM es relajado, es decir, la separación entre clases se realiza permitiendo errores de clasificación en los ejemplos de entrenamiento. El parámetro C define el margen de error en el que se pueden encontrar los ejemplos mal clasificados.
  - **NU_SVC:** Similar al tipo anterior. Permite la separación imperfecta de clases, utilizando en este caso el parámetro ν para fijar el límite de decisión.
  - **ONE_CLASS:** Este tipo de SVM requiere que todo el set de entrenamiento pertenezca a la misma clase. Así, la SVM crea una región de separación entre la clase y el resto del espacio de características.
  - **EPS_SVR:** Utilizada para problemas de regresión. La distancia entre los vectores de características del set de entrenamiento y el hiperplano debe ser menor que p. El parámetro C se utiliza para los ejemplos mal clasificados.
  - **NU_SVR:** También utilizada en regresión, similar a la anterior, se utiliza el parámetro ν en lugar de p.
- **Kernel = kernelType:** concreta el tipo de kernel de la SVM. Las posibilidades son las siguientes:
  - **LINEAR:** kernel lineal. La discriminación (o regresión según el caso) se realiza en el espacio de características original. Es la opción más rápida. $K(x_i, x_j) = x_i^T x_j$
  - **POLY:** kernel polinomial. $K(x_i, x_j) = (\gamma x_i^T x_j + coef0)^{degree}, \quad \gamma > 0$
  - **RBF:** kernel de función de base radial. Es considerada una buena elección en la mayoría de las aplicaciones. $K(x_i, x_j) = e^{-\gamma \|x_i - x_j\|^2}, \quad \gamma > 0$
  - **SIGMOID:** kernel sigmoideo. $K(x_i, x_j) = tanh(\gamma x_i^T x_j + coef0)$
  - **CHI2:** kernel exponencial chi^2 similar al kernel RBF. $K(x_i, x_j) = e^{-\gamma X^2(x_i, x_j)}, \quad X^2(x_i, x_j) = (x_i - x_j)^2/(x_i + x_j), \quad \gamma > 0$
  - **INTER:** kernel de intersección de histograma. Se trata de un kernel rápido. $K(x_i, x_j) = min(x_i, x_j)$
- **Grado = degree:** parámetro que marca el grado del polinomio del kernel polinomial.
- **Gamma = gamma:** parámetro gamma de las funciones kernel polinomial, RBF, sigmoidea y chi^2.
- **Coeficiente 0 = coef0:** coeficiente 0 de los kernel polinomial y sigmoideo.
- **C = Cvalue:** parámetro C del problema de optimización de la SVM. Utilizado en los tipos: C_SVC, EPS_SVR y NU_SVR.
- **Nu = nu:** parámetro ν del problema de optimización de la SVM que se utiliza con las SVM NU_SVC, ONE_CLASS, y NU_SVR.
- **P = p:** parámetro ϵ del problema de optimización de la SVM de tipo EPS_SVR.

- **Peso de clase = classWeights:** con la SVM de tipo C_SVC podemos asignar distintos pesos a cada clase. De esta forma, los pesos afectan a la penalización del error de clasificación en las distintas clases. A mayor peso, mayor penalización en los errores de clasificación de la clase correspondiente.
- **Criterio de finalización = termCrit:** criterio de finalización del proceso de entrenamiento iterativo de la SVM. Se puede especificar cierta tolerancia y/o el máximo número de iteraciones.

Una vez creada una SVM, se debe entrenar. El proceso es relativamente sencillo y se puede llevar a cabo mediante dos métodos distintos: *svm.train*, que se explica a continuación; y *svm.trainAuto*, que quedará explicado en [3.5.2]. Debido a que la SVM utilizada en un primer momento fue del tipo C_SVC con kernel lineal, por su facilidad de implementación y velocidad en ejecución, el método utilizado para el entrenamiento fue 3.4.

Fragmento de código 3.4: Método *svm.train*.

```
svm->train(trainingDataMat, ROW_SAMPLE, labelsMat);
```

Este método se compone de tres argumentos: dos son matrices y el otro un criterio para el tratamiento de esas matrices.

- **Muestras = trainingDataMat:** matriz en la que se deben encontrar los vectores de características extraídos del set de entrenamiento.
- **Criterio = ROW_SAMPLE:** define el criterio con el que se tratarán las matrices de entrenamiento. Existen dos formas de hacerlo:
  - **ROW_SAMPLE:** las muestras de entrenamiento están organizadas por filas en la matriz, cada fila una muestra. Éste es el criterio utilizado en esta aplicación.
  - **COL_SAMPLE:** en contraposición al anterior, las muestras de entrenamiento se organizan por columnas en la matriz.
- **Etiquetas = labelsMat:** matriz en la que se encuentran las etiquetas asociadas a las muestras de entrenamiento.

La clase SVM necesita que las matrices utilizadas en este método tengan una estructura y un tipo de dato concreto. Para ello se deben crear las matrices de la siguiente forma:

Fragmento de código 3.5: Adecuación del formato de las matrices de entrenamiento para su uso en *svm.train*.

```
Mat trainingDataMat(TOTAL, DESCRIPT, CV_32FC1, trainingData);

Mat labelsMat(TOTAL, 1, CV_32SC1, labels);
```

De esta forma, TOTAL hace referencia al número de columnas de la matriz, correspondiente a la cantidad de imágenes, y por tanto de descriptores que se usan en el set de entrenamiento. DESCRIPT define el número de filas de la matriz, que se corresponde con el número de componentes de cada descriptor o vector de características. El tercer parámetro fija el tipo de dato, exigido por *OpenCV* en esta aplicación. Por último, el cuarto parámetro indica de qué variable se extraen los datos para generar la nueva matriz. Como se puede observar en el fragmento de código, se debe hacer tanto para la matriz de características como para la de etiquetas.

3.3.2 Conclusiones

Como se mencionaba en 3.3, se encontraron ciertas dificultades a la hora de extraer los datos de entrenamiento de los ficheros de texto y guardarlos de forma ordenada en las matrices de entrenamiento. En primer lugar, porque al estar guardados en ficheros de texto planos, el bucle que los analizaba no garantizaba que cada fila de la matriz correspondiera perfectamente con un único vector de características, y la forma de observar esto no es inmediata debido a la cantidad de datos que se tratan. Además, debido también a esta gran cantidad de datos utilizados, distintos problemas de memoria se iban sucediendo. La ejecución se veía interrumpida por violaciones de segmento, es decir, intentos por parte del programa de acceder a zonas de memoria a las que no le está permitido acceder. Después de distintas pruebas y estudios, se consiguió solucionar estos problemas alojando previamente la memoria necesaria para cada matriz.

3.4 Algoritmo de Detección

Una vez entrenada una SVM con los datos extraídos del set de imágenes, el siguiente paso consistió en programar el algoritmo de detección. Este algoritmo se encarga de tomar la imagen que se le proporciona, cargar la SVM previamente entrenada, analizar la imagen para extraer sus descriptores y, gracias a la SVM, decidir cuáles pertenecen a la clase de media persona y marcar su localización sobre la imagen dibujando un rectángulo alrededor de cada detección.

Para implementar el algoritmo, se partió de un ejemplo que ofrece *OpenCV* para la detección de personas completas. El código en cuestión recibe el nombre de *peopledetect.cpp* y se estructura de la siguiente forma: el programa crea un objeto de la clase HOG con los parámetros por defecto, obtiene como vector soporte el detector de personas que la librería *OpenCV* tiene ya entrenado y listo para su utilización (se trata de la función *hog.getDefaultPeopleDetector*), lo carga en la SVM y analiza la imagen mediante el método *detectMultiScale* de la clase HOG [3.4.1]. Una vez analizada la imagen por completo, extrae las posiciones en las que se encontraron detecciones positivas y las enmarca para ofrecer al usuario una vista de la imagen en cuestión, con todas las personas que ha localizado enmarcadas en un rectángulo, como se puede observar en la figura 3.5.

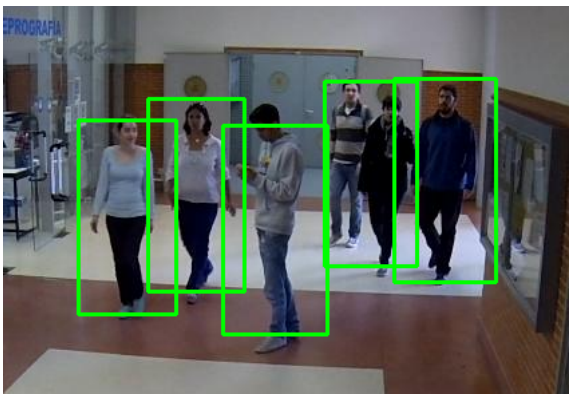

Figura 3.5: Detección de personas con el código de ejemplo implementado en *peopledetect.cpp*.

Gracias a lo práctico que resultaba este ejemplo y la sencillez a priori para modificarlo y adecuarlo a nuestra aplicación, se optó por trabajar sobre él. Las modificaciones que requería no eran demasiadas. En primer lugar, para mantener la coherencia con el resto de la aplicación y no provocar errores debido a los parámetros, se debía definir el objeto de HOG de la misma manera que se hizo en 3.2. En segundo lugar,

se hacía imprescindible, como es lógico, cargar la SVM entrenada para la aplicación; a fin de utilizar el vector soporte de la misma en el proceso de detección. A partir de ahí, el método *detectMultiScale* se encargaba de analizar la imagen y realizar las detecciones, que serían mostradas al finalizar el programa, de la misma manera que en el ejemplo de partida.

Cabe destacar la importancia de *detectMultiScale* como núcleo de este proceso. Al fin y al cabo es el método encargado de la búsqueda de formas en la imagen que se asemejen a las que se utilizaron para entrenar la SVM, es decir, las formas que buscamos, y por tanto, el objetivo de nuestra aplicación.

3.4.1 detectMultiScale en detalle

Tal y como se ha descrito, *detectMultiScale* es un método de la clase HOG encargado de analizar las imágenes que se le pasan como argumento. Su funcionamiento se basa en la extracción de descriptores HOG y el procedimiento de ventana deslizante [3]. Además, mediante el reescalado de la imagen, es capaz de realizar detecciones de los objetos aunque éstos aparezcan con distinto tamaño en la imagen.

A continuación se procede a explicar en detalle el funcionamiento de *detectMultiScale*, así como sus parámetros y la influencia de éstos en los resultados que se obtienen [26]:

Fragmento de código 3.6: Método *hog.detecMultiScale*.

```
hog.detectMultiScale(img, found, 0, Size(16,16), Size(0,0), 1.05, 2);
```

- **Imagen = img:** se trata de la imagen sobre la que se realizará la detección de objetos. Es capaz de trabajar con imágenes en color o en escala de grises.
- **Localizaciones = found_locations:** ésta es la variable en la que se guardan los datos de localización de los objetos detectados en la imagen. Guarda la coordenada de la esquina superior izquierda de la ventana en la que se realizó la detección.
- **Umbral de detección = hit_threshold 0:** representa el umbral de distancia entre las características y el hiperplano de clasificación de la SVM.
- **Desplazamiento de ventana = win_Stride Size (16,16):** fija en píxeles el desplazamiento, tanto en el eje x como en y, de la ventana de detección que recorre la imagen.
- **Relleno = padding Size (0,0):** parámetro que define cierta cantidad de píxeles añadidos en los bordes de la ventana de detección. Como ya se destacó en 3.2.1, se utiliza para mejorar las detecciones en los bordes de la imagen.
- **Escalado = scale0 1.05:** factor de escala aplicado a la imagen cada vez que se completa el análisis de cada escala.
- **Umbral de agrupación = group_threshold 2:** este coeficiente regula el umbral de similitud. Cuando los objetos son detectados en la imagen, puede darse el caso de que sean detectados varias veces por distintas ventanas. Este factor de similitud elimina ciertos rectángulos que se dibujarían en la imagen cuando el objeto ha sido detectado en múltiples ventanas.

Este método se basa en el cálculo de los descriptores de la imagen sobre la que se quieren detectar los objetos. Para ello, una ventana de detección recorre la imagen según el desplazamiento definido en la función. Esta ventana es del tamaño definido en la creación del objeto. De ahí la importancia de que el

HOG utilizado en la etapa de extracción de características para el entrenamiento y el utilizado en esta fase tengan los mismos parámetros. En cada posición de la ventana de detección, se calculan los descriptores de esa región y se pasa este vector a la SVM para ser analizado y clasificado. Si esta clasificación resulta positiva, se almacena la información de su posición relativa en la imagen, y la ventana se desplaza hasta la siguiente localización, repitiendo el proceso. Una vez la imagen es analizada completamente a una escala, se realiza el reescalado con el factor definido, obteniendo una nueva imagen de mayor tamaño, que vuelve a ser analizada mediante la ventana deslizante. De esta forma el método permite realizar detecciones de los objetos de interés en planos alejados de la imagen, ya que los aumenta y así pueden ser detectados. Al finalizar el proceso en todas las escalas definidas, todas las clasificaciones positivas que ha entregado la SVM representan las detecciones que nos interesan. Debido al reescalado, cabe la posibilidad de que un objeto sea detectado en distintas escalas, por lo que el algoritmo realiza una agrupación de las detecciones del mismo objeto para mostrarlas como una única detección.

Una vez explicados tanto los parámetros como el funcionamiento, cabe destacar dos parámetros de *detectMultiScale* cuya importancia es fundamental. El ajuste correcto de estos valores, no sólo con respecto a los objetos de HOG creados previamente, sino también con respecto a la imagen que va a ser analizada, tiene tremendas implicaciones tanto en la exactitud del detector como en la velocidad de computación.

El primero de ellos es el parámetro que define el desplazamiento de la ventana. Si el desplazamiento definido es muy pequeño, el análisis de la imagen será mucho más exhaustivo y por tanto será más probable que no se pierdan detecciones. Sin embargo, la velocidad de ejecución del algoritmo se ve muy mermada, ya que se deben calcular muchos más descriptores en todas las escalas de la imagen. Por el contrario, un desplazamiento de ventana más amplio incrementará la velocidad del algoritmo, pudiendo provocar en cambio que ciertas detecciones que deberían haberse realizado se pasen por alto.

El segundo parámetro de gran importancia es el del factor de escalado, y el compromiso que impone es similar al anterior. Como ya se ha mencionado, este factor define el tamaño que tendrá la imagen en cada iteración del algoritmo. Por tanto, un factor de escala pequeño permitirá analizar la imagen en muchas escalas, comprometiendo el tiempo de ejecución para obtener mejores detecciones en distintos planos de la imagen. Del mismo modo, un factor de escala más grande reducirá el tiempo de ejecución pero aumentará las posibilidades de perder detecciones de objetos situados en zonas alejadas de la imagen. Se debe tener en cuenta que para cada reescalado de la imagen, el método de ventana deslizante se ejecuta por completo, por lo que los efectos de modificar este parámetro de escala influyen enormemente en el tiempo computacional del sistema; ya que se deben analizar, extraer descriptores y pasarlos a la SVM para la clasificación en todas las ventanas de todas escalas.

3.4.2 Conclusiones

Siendo ésta la última fase del sistema de detección, es en este punto donde reparamos en la importancia de haber mantenido la coherencia en cuanto a funciones y parámetros que han sido utilizados durante el resto del proceso. Manteniendo dicha coherencia, los errores se minimizaban, además de que resultaba más sencilla la programación y la realización de las distintas pruebas.

Por otra parte, interpretar correctamente como afectan los parámetros de escalado y desplazamiento de ventana en el proceso de detección, y establecer éstos en concordancia con la imagen que se analizaba y su tamaño, permitió ahorrar tiempo en la fase de pruebas cuando las imágenes a analizar eran de un tamaño considerable.

3.5 Pruebas y modificaciones

Una vez implementado un sistema completamente funcional, se procedió a probar su eficiencia mediante distintas imágenes, tanto extraídas del set de test de INRIA, como imágenes de prueba y frames de vídeos utilizados en el trabajo de referencia [3]. Gracias a estas pruebas y debido a los resultados negativos que ofrecían, se estudió más a fondo el funcionamiento interno de las funciones y del propio sistema. Se evaluó la viabilidad del algoritmo y se detectaron distintos problemas que se detallan a continuación.

3.5.1 Primer problema y solución propuesta

El primer gran problema que se detectó a la hora de realizar pruebas fue que, tanto en imágenes del set de test como en otros tipos de imagen, incluso aquellas que no contenían personas, el sistema dibujaba en el centro de la imagen un recuadro de 64x64 píxeles. Independientemente de los parámetros que se modificaran, la respuesta siempre resultaba la misma, como se puede ver en la figura 3.6. Se cambiaron los parámetros de HOG en la extracción de descriptores, se probaron distintos parámetros de C en la SVM lineal, e incluso se llegó a modificar el set de imágenes de entrenamiento, pero el problema persistía.

Figura 3.6: Primeros resultados obtenidos.

Mediante un análisis mucho más exhaustivo del código en todas sus fases, se observó que el problema residía en el vector soporte de la SVM. Se comprobó que efectivamente la variable que almacenaba este vector quedaba rellena y con datos aparentemente correctos en la fase de entrenamiento. Sin embargo, a la hora de utilizarlo en la fase de detección, el mismo vector quedaba vacío y *detectMultiScale* trabajaba sin un vector soporte de referencia. Tras un proceso de búsqueda y análisis de la documentación disponible, se descubrió que la forma en la que se estaban guardando los descriptores y el vector soporte para utilizarlos en las fases siguientes no era óptima, y se producían importantes pérdidas de información. La solución implementada fue cambiar el almacenamiento de los datos en ficheros .txt de texto planos por archivos con formato .yml. Para estos archivos, a parte de tener una estructura propia y un formato definido, existen funciones en C de lectura y escritura que aseguran que la información se almacena correctamente y se guarda en las variables de interés sin pérdida de información. Con este cambio tanto en la fase de extracción de descriptores como en la de entrenamiento de la SVM, donde se podían guardar y acceder sin problemas tanto a los descriptores como al modelo creado para la máquina de soporte vectorial; se empezaron a obtener resultados distintos, aunque no correctos.

3.5.2 Segundo problema y solución propuesta

Ante la obtención de resultados erróneos, es decir, la incapacidad del sistema para realizar detecciones de la parte superior de las personas que aparecen en las imágenes; se siguió investigando en qué puntos el sistema podría estar fallando. Se realizaron tres cambios importantes que influyeron enormemente en los resultados que se venían obteniendo.

El primero fue el cambio en el set de imágenes de entrenamiento. Se descubrió en internet cierta información relevante que afectaba a toda la aplicación: *OpenCV* no está optimizado para utilizar HOG con otros tamaños de ventana que no sean de 64x128, tamaño original utilizado por Navneet Dalal [5]. Además, un error importante y que no se había tenido en cuenta hasta el momento, era que en el set de imágenes de entrenamiento es importante que el número de muestras negativas sea mucho mayor que el de las positivas. La solución pasó por crear un set de imágenes completamente nuevo (ver figura 3.7). Para las muestras positivas se extrajeron mediante un programa de procesado de imágenes los recortes que encuadraban la mitad superior de una persona. Estos recortes se obtuvieron de distintas imágenes de la base de datos de INRIA, hasta llegar a un total de 414, posteriormente redimensionadas a 64x128. Para las imágenes negativas se realizó un pequeño script que obtenía las imágenes negativas, también de INRIA; las dividía en 4 parches, definidos por la mitad superior dividida en izquierda y derecha, y la mitad inferior de la imagen, también dividida en izquierda y derecha. Estos parches se redimensionaban al tamaño requerido haciendo un total de 8520 muestras negativas. De esta manera se solucionó el problema de las dimensiones necesarias para el correcto funcionamiento de HOG en *OpenCV*, así como el de las proporciones entre muestras positivas y negativas que se deben cumplir para optimizar el funcionamiento de la SVM.

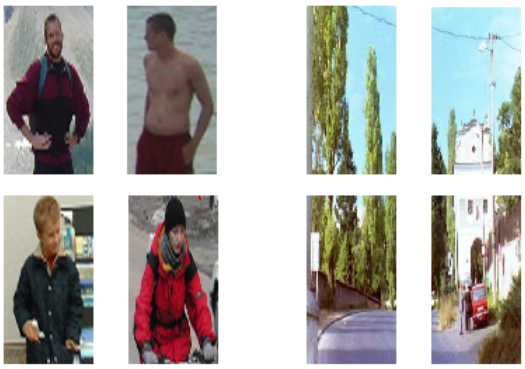

Figura 3.7: Visualización de algunas de las imágenes positivas y negativas del nuevo set.

El segundo cambio que se realizó fue en el kernel de la SVM. Debido a la cantidad y dimensión de los datos con los que se trabajaba, se planteó el uso del kernel RBF para la aplicación. La definición de la nueva SVM no difería prácticamente de la SVM lineal, como se puede ver comparando los fragmentos de código de cada una, 3.3 y 3.7.

Fragmento de código 3.7: Definición de un objeto de SVM, esta vez con kernel RBF.

```
Ptr<SVM> svm = SVM::create();
svm->setType(SVM::C_SVC);
svm->setKernel(SVM::RBF);
svm->setTermCriteria(TermCriteria(TermCriteria::MAX_ITER, 1000, 1e-6));
```

Los cambios importantes llegaban a la hora de entrenar la máquina. Debido a que los valores gamma y C influyen de forma notable en los resultados que se obtienen, y que el rango de valores que éstos pueden tomar es bastante amplio, se hizo indispensable el estudio y posterior utilización del método *trainAuto*.

Fragmento de código 3.8: Definición de distintos parámetros necesarios para el correcto uso del método *svmtrainAuto*.

```
Ptr<TrainData> autoTrainData = TrainData::create(trainingDataMat, ROW_SAMPLE,
    labelsMat);

ParamGrid Cgrid = SVM::getDefaultGrid(SVM::C);
ParamGrid gammaGrid = SVM::getDefaultGrid(SVM::GAMMA);
ParamGrid pGrid = SVM::getDefaultGrid(SVM::P);
pGrid.logStep = 1;
ParamGrid nuGrid = SVM::getDefaultGrid(SVM::NU);
nuGrid.logStep = 1;
ParamGrid coeffGrid = SVM::getDefaultGrid(SVM::COEF);
coeffGrid.logStep = 1;
ParamGrid degreeGrid = SVM::getDefaultGrid(SVM::DEGREE);
degreeGrid.logStep = 1;

svm->trainAuto(autoTrainData,10, Cgrid, gammaGrid, pGrid, nuGrid, coeffGrid,
    degreeGrid, false);
```

Como se puede observar en el fragmento de código 3.8, la matriz de entrenamiento se crea de manera muy similar al caso de la SVM lineal a partir de las matrices de características y etiquetas; con la pequeña diferencia de que en este caso es necesario que el tipo de variable sea *TrainData*. La mayor diferencia reside en la creación de las rejillas (*grids*) de datos. Éstas se pueden definir por defecto gracias a las funciones de *OpenCV*, y marcan el aumento del respectivo parámetro en cada iteración del proceso de entrenamiento. Cabe destacar que las líneas que definen el *logStep* se utilizan para indicar al algoritmo que el parámetro con el que se asocia no debe ser optimizado. Como se puede observar en 3.8, ya que se utiliza una SVM del tipo C_SVC con kernel RBF, los parámetros que influyen en el proceso, y por tanto los que deben ser optimizados en el entrenamiento son gamma y C.

Para entender el proceso de entrenamiento es importante explicar los parámetros de la función *trainAuto* [29]:

- **Datos = data autoTrainData:** se trata de la matriz que almacena los datos de entrenamiento, es decir, características y etiquetas.
- **Parámetro de validación cruzada = kFold 10:** este parámetro define el número de veces que se divide el set en subconjuntos. Uno de ellos es utilizado para testear el modelo, mientras que los otros se utilizan para entrenarlo. De esta forma, el algoritmo que define la SVM se realiza internamente el número de veces que este valor determine.
- **Rejillas = Grid:** explicadas anteriormente. Cada rejilla marca el valor en la siguiente iteración del respectivo parámetro.
- **Balanceo = balanced false:** define la forma en la que se crean los subconjuntos y las proporciones que tienen unos respecto a otros.

Una vez explicados los parámetros de *trainAuto* y remarcada la importancia de gamma y C en las SVM de kernel RBF, es importante también dar una idea intuitiva de cómo afectan estos parámetros al proceso de entrenamiento y a los resultados obtenidos. Gamma define hasta dónde puede afectar la influencia de un sólo ejemplo de entrenamiento. Con valores pequeños de este parámetro permitimos que la influencia sea mayor, y viceversa. Entonces, gamma puede ser interpretado como la inversa del radio de influencia que tienen las muestras seleccionadas por el modelo como vectores soporte. El parámetro C

representa el compromiso entre permitir la clasificación errónea de algunos ejemplos y la simplicidad que se pueda obtener en la superficie de decisión. De esta forma, valores bajos de C suavizan esta superficie, mientras que valores más altos tratan de clasificar todos los ejemplos correctamente, dando al modelo cierta libertad para seleccionar más muestras como vectores soporte.

Con todo esto, llegamos a la tercera y última modificación que se realizó en el sistema. Tras distintas búsquedas de información, se descubrió que el método *hog.setSVMDetector*, que se utiliza para proveer al detector del vector soporte necesario, requiere que este vector tenga una forma específica. En concreto, requiere extraer *rho* de la función de decisión de la SVM, e introducirlo en el vector soporte como el último de los valores del vector. Para ello se obtuvo un pequeño script que implementaba este proceso y se añadió al código en el que se realiza la detección [30].

3.5.3 Resultados tras las últimas modificaciones y conclusiones

Una vez realizadas todas las modificaciones descritas en la sección anterior, se procedió a realizar más pruebas con las distintas imágenes para comprobar si el proceso de detección había mejorado como se esperaba. Los primeros resultados invitaban al optimismo, hasta que se comprobó que el sistema parecía más bien repetir cierto patrón que realizar detecciones correctas, como se puede observar en la figura 3.8.

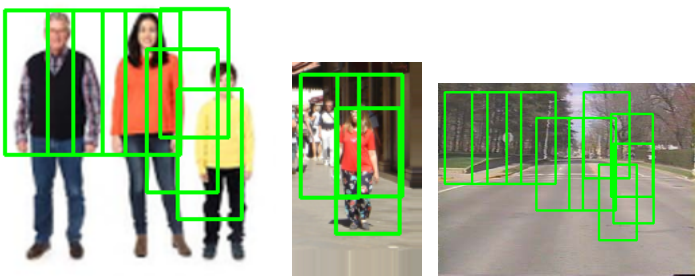

Figura 3.8: Resultados obtenidos después de las modificaciones.

Además, los resultados que proporcionaba el sistema ante imágenes semejantes al de una aplicación real (figura 3.9 y 3.10) llevaban a dos conclusiones distintas. La primera era la posibilidad de que el sistema tuviera errores aún no detectados que estuvieran haciendo fallar todo el proceso. La segunda conclusión, a la que se llegó finalmente después de volver a estudiar e investigar todo el código de la aplicación, era la de que *OpenCV* y su implementación de HOG, con todas las funciones que conlleva, no están optimizados o no son suficientes para desempeñar la función de detección de media persona que se ha tratado de implementar en este proyecto. Aunque a priori, y teóricamente no hay nada que niegue la posibilidad de realizar este sistema, los resultados no hacen sino evidenciar la falta de exactitud del detector, imposible de asumir para la aplicación en un sistema real.

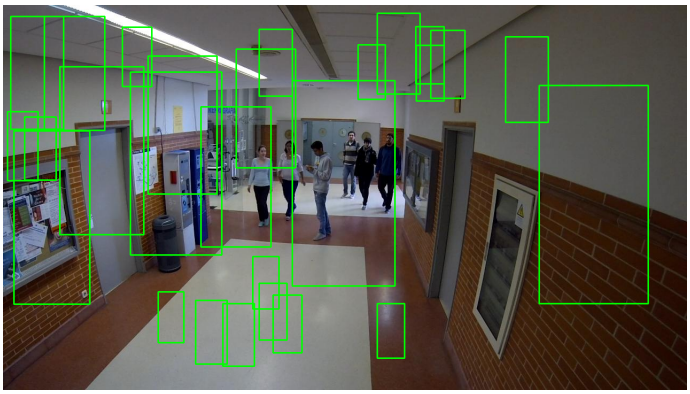

Figura 3.9: Resultados obtenidos después de las modificaciones en imágenes de aplicación real.

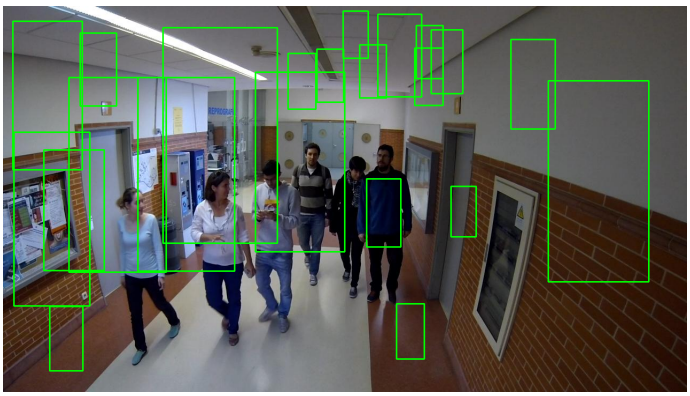

Figura 3.10: Resultados obtenidos después de las modificaciones en imágenes de aplicación real.

Con todos estos resultados sobre la mesa, siendo conscientes de la incapacidad de este sistema para desempeñar la función para la que había sido programado, se decidió abordar el algoritmo mediante *Matlab*. Dadas las facilidades que ofrece este lenguaje para el tratamiento de matrices, la implementación de HOG y de las SVM, y gracias al conocimiento adquirido sobre el algoritmo de detección a lo largo de la programación en *OpenCV*; se consideró que la realización de este sistema en *Matlab* no consumiría un tiempo excesivo del proyecto y tal vez se consiguieran los resultados esperados.

3.6 Implementación en Matlab

Mediante el cambio de lenguaje de programación a *Matlab* se consiguieron resultados relativamente rápido, aunque no fueron mejores que los conseguidos con *OpenCV*. El algoritmo seguía, como es lógico, las mismas fases y el mismo orden. Sin embargo, al ser más inmediata la implementación de HOG y de la SVM, la fase de extracción de descriptores para el entrenamiento de una máquina de soporte vectorial junto con el propio entrenamiento de la misma se realizaban dentro del mismo script, consiguiendo un modelo de SVM que se utilizaría en la etapa de detección. Por otra parte, el algoritmo de detección era más complicado de implementar, ya que *Matlab* carece de funciones para el análisis de imagen mediante el proceso de ventana deslizante y el reescalado de la imagen; por lo que se hacía imprescindible implementar de alguna manera este tipo de análisis. El problema de realizar este proceso mediante bucles que recorren la imagen, como se planteó en este proyecto, reside en la velocidad de ejecución. Al no utilizar funciones completas ya creadas en *Matlab*, el proceso es mucho más lento ya que no está optimizado.

3.6.1 Extracción de descriptores HOG y entrenamiento de la SVM

Programar el proceso de extracción de descriptores y el entrenamiento de la SVM era mucho más sencillo y cómodo que su implementación en *OpenCV*. Las funciones *extractHOGFeatures* y *fitcsvm* agilizaban en gran medida esta tarea. El único requisito para realizar el proceso era la necesidad de organizar en carpetas del directorio de trabajo las imágenes que se usan para el entrenamiento, de tal manera que en una carpeta se encontraran las subcarpetas con imágenes de entrenamiento positivas y negativas separadas. Así, la función *imageDatastore* podía acceder a la carpeta principal y organizar el set de entrenamiento etiquetando automáticamente las imágenes con el nombre de la carpeta a la que pertenecían. Esto se consigue utilizando la pareja de argumentos *LabelSource* y *foldernames* que se puede observar en la línea 2 del código que implementa la función 3.9.

Una vez organizado el set, era necesario crear ciertas variables para realizar el proceso de extracción. Se estableció en primer lugar la matriz en la que se guardarían los vectores de características. Para ello se necesitaba saber las cantidad de imágenes que se utilizan en el set y la longitud del vector de características. Ambos valores eran inmediatos. El primero porque se podía extraer como un argumento de la variable creada por *imageDatastore*. El otro por el conocimiento adquirido sobre HOG y los resultados que produce con un tamaño de ventana de 64x64 y los tamaños de bloque y celda, número de contenedores del histograma y demás valores establecidos por defecto. También se definió la ventana de la que se extraen los descriptores como una región de interés de la imagen sobre la que se realiza el proceso de extracción de características. Con todo esto, se implementó un bucle que recorría el set de entrenamiento extrayendo los descriptores de cada imagen y guardándolos en la variable correspondiente (*trainingFeatures*). Mediante esta variable y la que almacenaba las etiquetas, se entrenaba una SVM gracias a la función *fitcsvm*. Como se puede ver en el código de esta fase [3.9], la SVM se entrenaba con los parámetros por defecto que define Matlab y que se ajustaban a nuestra aplicación. El kernel lineal, el método de optimización y el número de iteraciones, que no se especifican y por tanto se definen con sus valores por defecto, se hacían suficientes para realizar el proceso.

Fragmento de código 3.9: Código que implementa la extracción de descriptores y el entrenamiento de la SVM en *Matlab*.

```
1  function svm = extract_and_train
2      trainset = imageDatastore   ('Train','IncludeSubfolders',true,'LabelSource','
           foldernames');
3      numImages = numel(trainset.Files);
4      HogFeatureSize = 1764;
5      trainingFeatures = zeros(numImages, HogFeatureSize, 'single');
6      p1 = [1,1];
7      p2 = [64,64];
8      p0 = [p1;p2];
9
10   for i = 1:numImages
11     img = readimage(trainset,i);
12     cimg = imcut(p0,img);
13     trainingFeatures(i,:) = extractHOGFeatures(cimg);
14   end
15
16     trainingLabels = trainset.Labels;
17     svm = fitcsvm (trainingFeatures, trainingLabels);
18 end
```

Una vez ejecutado el código, el modelo de la SVM obtenido se podía guardar y de esta forma ser utilizado en los distintos procesos cargándolo desde el directorio de trabajo en el espacio de variables.

3.6.2 Algoritmo de Detección

Para realizar el algoritmo de detección e implementar el método de ventana deslizante junto con el reescalado de la imagen, se empezó por buscar información en internet que ayudara a plantearlo. Se encontró un código utilizado para la detección de caras en imágenes [31], que sirvió de gran ayuda aunque no contemplase el reescalado de las imágenes para realizar las distintas detecciones. Primero se estudió y comprobó el funcionamiento del mismo para entender el proceso, las distintas fases involucradas y el orden en que se debían realizar. Gracias a este código y a ciertas partes del mismo que se utilizaron, siendo modificadas para adecuarlas a nuestra aplicación, se pudo implementar primero el algoritmo de detección basado en el método de ventana deslizante. Posteriormente se incluyó la funcionalidad de la detección multiescala escribiendo un nuevo script que incluía la función anterior.

La primera función, a la que se dió el nombre de *detect_in_image*, y de la que podemos ver su código a continuación (3.10), necesitaba como argumentos la imagen a analizar, el modelo de la SVM que se utilizaría para decidir si la detección es positiva o negativa, y el tamaño de ventana de detección, que en nuestra aplicación debía coincidir con el utilizado en el código de extracción y entrenamiento (64x64). A continuación se procede a explicar en detalle el código de la función 3.10.

Fragmento de código 3.10: Código que implementa la detección de objetos de una imagen en *Matlab*.

```
function [detections, score, boxPoint] = detect_in_img (img, model, wSize)
    topLeftRow = 1;
    topLeftCol = 1;
    [bottomRightCol, bottomRightRow, ~] = size(img);
    wStride_y = round(wSize(1)/8);
    wStride_x = round(wSize(1)/8);
    fcount = 1;

  for y = topLeftCol:wStride_y:bottomRightCol-wSize(2)+1
    for x = topLeftRow:wStride_x:bottomRightRow-wSize(1)+1
        p1 = [x,y];
        p2 = [x+(wSize(1)-1), y+(wSize(2)-1)];
        po = [p1; p2];
        crop = imcut(po,img);
        featureVector{fcount} = extractHOGFeatures(crop);
        boxPoint{fcount} = [x,y];
        fcount = fcount+1;
    end
  end

  for k= 1:length(featureVector)
    [detections{k}, score{k}] = predict (model, featureVector{k});
  end
end
```

Las primeras variables que se definen se utilizan para extraer el tamaño de la imagen, que será clave a la hora de definir los dos bucles anidados que implementan la ventana deslizante. Después, se define el

desplazamiento de la ventana de detección, dado en relación al tamaño de la propia ventana, argumento requerido en la definición de la función. De esta forma se evitan los problemas que se podrían dar a la hora de recorrer la imagen y que el desplazamiento fuera demasiado grande, de tal forma que la ventana saliera de la imagen y provocara un error en ejecución. A continuación, la definición de una variable auxiliar para la iteración de los bucles, y la definición de los propios bucles.

En la cabecera de los bucles se definen, tanto para el eje X como para el eje Y, el punto en el que se ubicará la esquina superior izquierda de la ventana de detección en cada iteración del bucle, a la par que se define cuánto se moverá esta ventana en función del desplazamiento definido anteriormente. Utilizando el tamaño de ventana pasado como argumento, se define la región de interés, recortada mediante la función *imcut*, de la que se extraerá y almacenará su descriptor HOG. También resulta de vital importancia almacenar para cada ventana el punto de la esquina superior izquierda, ya que posteriormente se utilizará para dibujar los rectángulos que encuadran las detecciones positivas. Por último, se el bucle termina después de aumentar en uno la variable auxiliar que define la posición en la que se guardan los datos, tanto en la matriz de características, como en la de los puntos que representan cada ventana de detección. El siguiente bucle, con tantas iteraciones como número de descriptores extraídos, recorre la matriz de características y realiza para cada descriptor la predicción de su clase mediante el método *predict*. Este método es propio de las SVM y utiliza el modelo que se le pasa como argumento. Para terminar, la función *detect_in_image* devuelve la clase a la que pertenece cada detección; su puntuación, es decir, un valor que define cómo de probable es que la detección pertenezca a la clase; y el punto de la esquina superior izquierda que relaciona cada detección con su ventana.

Una vez comprobado el correcto funcionamiento de la función *detect_in_image*, se podía trabajar en la implementación de la detección multiescala. Para ello se escribió otra función a la que se dió el nombre de *detect_multiscale* [3.11], que sirviéndose de la anterior función y de un nuevo bucle, permitía realizar las detecciones de los objetos que aparecían con menor tamaño en la imagen ya que se encontraban más lejos.

Fragmento de código 3.11: Código que implementa la detección multiescala en *Matlab*.

```
function detect_multiscale (image, model)
    imagen = imageDatastore (image);
    img = readimage(imagen,1);
    scale_factor = 0.2;
    wSize = [64,64];

  for scale = 1:scale_factor:1.8
    Res_img = imresize(img,scale);
    [detections, score, boxPoint] = detect_in_img (Res_img, model, wSize);
  end

    posdetect = find([detections{:}] == 'pos');
    bBox = cell2mat(boxPoint(posdetect));
    figure(1); cla;
    imshow (img);
    hold on;

  for N = 1:length(posdetect)
    rectangle('Position',[bBox(N),bBox(N+1),wSize],'LineWidth',1, 'EdgeColor','r');
  end
end
```

Como se puede observar en la primera línea del código 3.11, la función *detect_multiscale* no devuelve ningún valor, sino que se centra en obtener las detecciones de los objetos para los que se ha entrenado el modelo que se le pasa como parámetro, y dibujar los rectángulos que representan esas detecciones en la imagen de interés, la cual también se le debe pasar como argumento.

Lo primero que hace la función es leer la imagen en cuestión y establecer los dos parámetros de mayor importancia en la detección, factor de escala y desplazamiento de ventana, explicados en 3.4.1. El siguiente paso consiste en un bucle cuyas iteraciones vienen definidas por este factor de escala. En cada iteración del bucle se redimensiona la imagen y se obtienen todas las detecciones, ya sean de clase positiva o negativa, su puntuación y su localización en la imagen mediante la función *detect_in_image* explicada anteriormente. Una vez terminado el bucle se habrán analizado y extraído los descriptores de toda la imagen en todas las escalas, por lo que el siguiente paso consiste guardar en un vector únicamente las detecciones que pertenecen a la clase positiva, ya que son las que nos interesan para la aplicación. De igual forma, se deben guardar todas las posiciones de estas detecciones positivas. Por último, se muestra la imagen, y un bucle con tantas iteraciones como detecciones positivas se han encontrado dibuja sobre la imagen el recuadro que engloba cada objeto detectado por el sistema.

3.6.3 Resultados obtenidos

Una vez implementado todo el proceso, se procedió a comprobar qué resultados se obtenían en imágenes similares a las que se utilizaron en el sistema de *OpenCV*. El tiempo que necesitaba el algoritmo para obtener y mostrar los resultados era demasiado largo, imposible de ajustar a las necesidades de un sistema en tiempo real. Además, como es obvio, con las imágenes de prueba extraídas de secuencias de vídeo del trabajo de referencia [3], el tiempo de computación se incrementaba muchísimo ya que las imágenes eran mucho más grandes que las que se utilizaron para realizar las primeras pruebas.

Ya en las primeras imágenes que se utilizaron se evidenciaba la incapacidad del sistema para realizar las detecciones de la mitad superior de las personas que aparecen en ella, de la misma forma que ocurría con la implementación mediante C++ y *OpenCV*, como se puede ver en la siguiente figura.

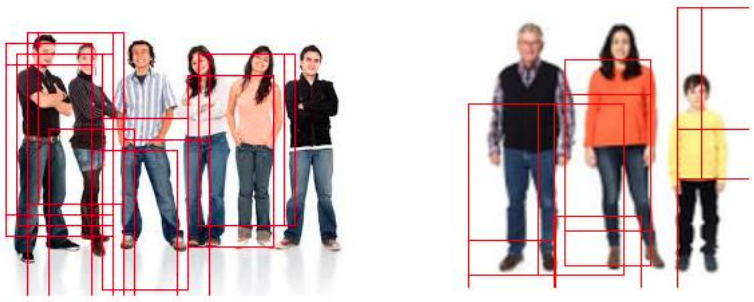

Figura 3.11: Resultados obtenidos con el sistema implementado en Matlab.

Los resultados obtenidos mediante *Matlab* en las imágenes extraídas de las secuencias de vídeo se asemejaban a los obtenidos con el primer sistema (figuras 3.9 y 3.10), en las que se observa una nube de recuadros que representarían las detecciones, pero que no coinciden con ninguna de las personas que aparecen en la imagen.

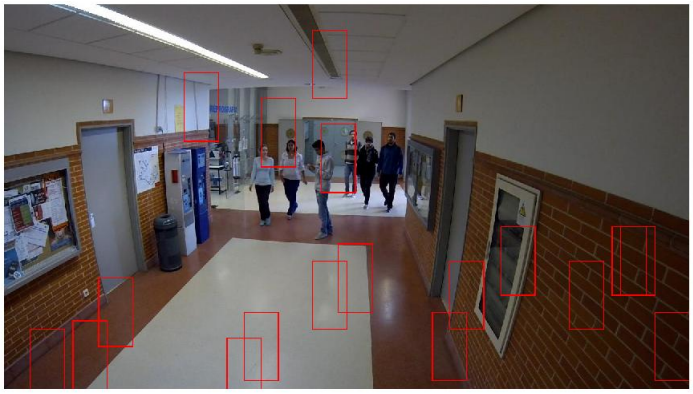

Figura 3.12: Resultados obtenidos sobre imágenes de aplicación real con el sistema implementado en Matlab.

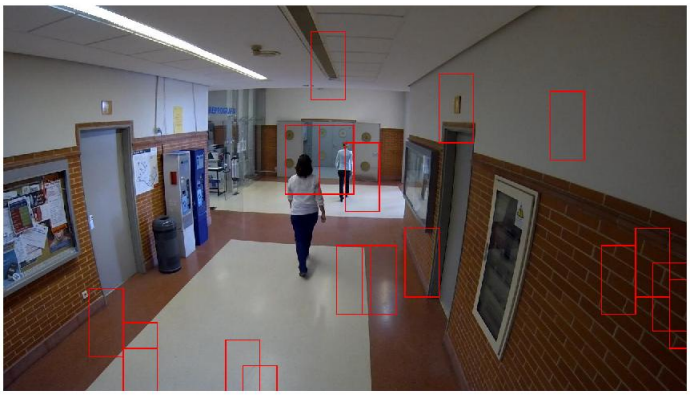

Figura 3.13: Resultados obtenidos sobre imágenes de aplicación real con el sistema implementado en Matlab.

Una vez obtenidos estos resultados, se modificaron los parámetros del factor de escala y el desplazamiento de ventana para realizar un análisis más exhaustivo de la imagen. El tiempo de ejecución se incrementaba considerablemente, y además no se conseguían mejores resultados. Incluso se empeoraba la calidad de las detecciones obteniendo una mayor cantidad de falsos positivos.

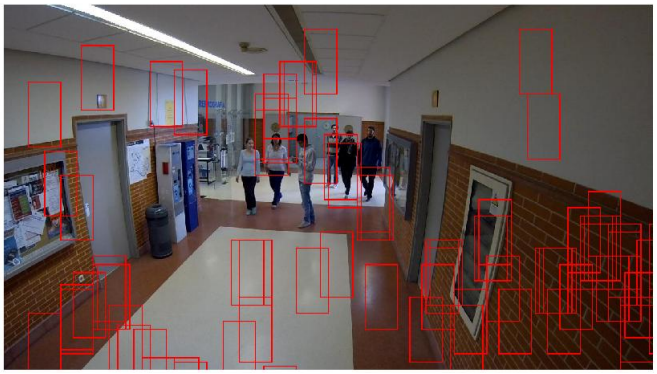

Figura 3.14: Resultados obtenidos sobre imágenes de aplicación real con el sistema implementado en Matlab y el cambio de parámetros mencionado.

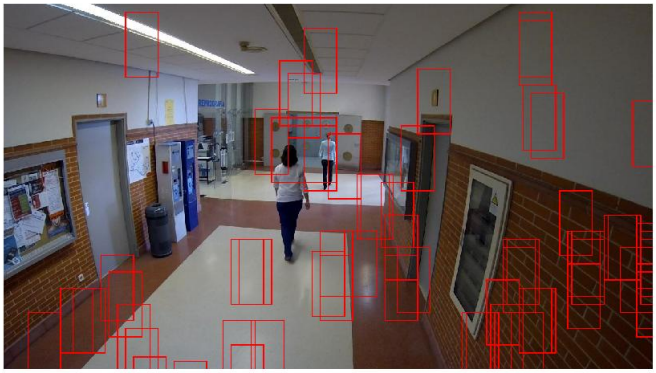

Figura 3.15: Resultados obtenidos sobre imágenes de aplicación real con el sistema implementado en Matlab y el cambio de parámetros mencionado.

Capítulo 4

Conclusiones y líneas futuras

En este capítulo se expondrán las conclusiones extraídas tras haber realizado y analizado el trabajo aquí desarrollado. A partir de estas conclusiones se indicarán algunas líneas futuras de trabajo que puedan seguirse para conseguir los resultados deseados.

4.1 Conclusiones

En este trabajo de fin de grado se ha implementado y evaluado un algoritmo para la detección de la mitad superior de las personas, con el objetivo de mejorar un sistema de detección previamente desarrollado en el caso de oclusiones parciales. El sistema desarrollado consta de una etapa de extracción de descriptores HOG, y un clasificador SVM entrenado para la detección de medio cuerpo.

Una vez implementada las fase de extracción de descriptores, y haciendo hincapié en que en un primer momento los datos se almacenaban en un fichero .txt; dada la cantidad de datos que se trataban, era de vital importancia el orden en el que éstos se guardaban así como el orden en el que se extraían y utilizaban en la fase de entrenamiento. Al manejar tal cantidad de datos, se detectaron problemas a la hora de extraerlos en la fase de entrenamiento. Con la utilización de ficheros .yml que se implementó posteriormente, se aseguraba que los datos se guardaban y utilizaban de forma correcta, ya que este tipo de archivos poseen un formato concreto y existen funciones de C++ dedicadas a la lectura y escritura de los mismos. Todo ello facilita el proceso y asegura que no haya errores en el orden de lectura. En conclusión, resulta más fácil y se consigue evitar más errores cuando se trabaja con ficheros con un formato determinado, preparados para la lectura y escritura mediante comandos predeterminados.

Otro punto a tener en cuenta reside en la importancia que adquiere el correcto uso de parámetros a la hora de definir los objetos de HOG utilizados, así como las funciones de entrenamiento de la SVM o del algoritmo de detección. Mantener la coherencia entre los parámetros de HOG utilizados a lo largo de las fases de extracción y detección hace que se mantengan las dimensiones de celdas, bloques, ventanas, descriptor, etc; evitando así errores tanto de programación como de ejecución, y manteniendo la coherencia en los resultados que se obtienen.

Además, conocer la influencia de ciertos parámetros en el proceso facilitaba la realización de pruebas y reducía el tiempo que éstas tomaban. El correcto ajuste de parámetros como el escalado o el desplazamiento de ventana en el algoritmo de detección permitía jugar con el tiempo que necesitaba para analizar la imagen y la calidad de las detecciones que se obtenían, agilizando la fase de pruebas.

En resumen, la forma de trabajar a lo largo de un proyecto de estas características, el estudio de las herramientas y algoritmos utilizados, así como la forma de organizarse y la forma de mantener la

coherencia a lo largo de todo el proceso adquiere gran importancia y, si se lleva a cabo de buena manera, facilita en gran medida tanto la programación de la aplicación como la redacción de la propia memoria. También se llega a la conclusión de la importancia de todos los conocimientos adquiridos y la capacidad de aplicarlos en distintas situaciones con distintas necesidades, haciendo referencia en especial al momento en el que se tuvo que comenzar a programar el sistema en *Matlab*. Habiendo estudiado y trabajado los algoritmos en *OpenCV*, resultó mucho más fácil conseguir el mismo sistema en otro lenguaje que si se hubiera comenzado directamente en *Matlab*.

Otra conclusión alcanzada después de terminar la programación tiene un carácter más personal, aplicable también al ámbito profesional. Los resultados conseguidos en este proyecto no son a los que se esperaba llegar cuando se planteó el mismo y se definieron las líneas de trabajo. Teóricamente no existía a priori ningún indicio de que la aplicación no pudiera ofrecer resultados correctos. Sin embargo, después de mucho estudio, análisis, trabajo y revisiones, los resultados sólo evidenciaban que mediante estos algoritmos no se podían conseguir resultados satisfactorios. Llegar a esa conclusión puede ser decepcionante, pero el camino tomado, aunque no haya llegado a buen puerto, sirve para que la investigación no vuelva a escoger el mismo recorrido. A veces nuestro trabajo, por duro que haya sido, no obtiene los resultados esperados, pero sirve para que otras personas avancen por diferentes ramas, y de esta forma se consigan los avances en investigación que estamos buscando.

4.2 Líneas futuras

Una vez concluído el proyecto, y ya que no se han conseguido resultados satisfactorios, algunas líneas de trabajo que se podrían seguir en el futuro para conseguir los objetivos de este trabajo pueden ser:

- **Utilización de otro tipo de descriptores:** en este proyecto se ha trabajado con los descriptores HOG por su conocida utilidad en tareas de detección de personas. Sin embargo, al estar trabajando con la parte superior de una persona como objeto de la detección, es posible que otro tipo de descriptores puedan ofrecer mejores resultados.

- **Utilización de otro tipo de clasificador:** aunque las máquinas de soporte vectorial apuntaban a ser el clasificador más indicado para esta aplicación, cabe la posibilidad de que distintos tipos de clasificadores, aunque sean más complejos, cumplan con la tarea que aquí se ha tratado de completar.

- **Combinación de técnicas de procesamiento de imagen:** es posible también que el método de ventana deslizante, que ofrece un análisis muy exhaustivo de la imagen, no sea el óptimo en tareas de detección en cuanto a tiempos de ejecución se refiere. Otras técnicas como la búsqueda selectiva mediante redes neuronales de convolución para determinar regiones de interés en la imagen podrían solventar estos problemas.

Bibliografía

[1] J. Gonzalez and V. Hernando, *Redes neuronales artificiales: fundamentos, modelos y aplicaciones*, ser. Serie Paradigma. Addison-Wesley Iberoamericana, 1995.

[2] C. Losada, J. J. Garcia, M. Mazo, J. Ureña, Á. Hernández, M. J. Díaz, and C. D. Marziani, "Uso de la técnica pca para la validación de la detección de objetos en entornos ferroviarios." in *Seminario Anual de Automatica, Electronica Industrial e Instrumentacion (SAAEI06)*, 09 2006, pp. 1–6.

[3] M. B. Ríos, "Detección y caracterización de personas en espacios inteligentes con cámaras de color," Ph.D. dissertation, Universidad de Alcalá de Henares (UAH). Escuela Politécnica Superior (EPS). Departamento de Electrónica, 2015.

[4] "The origins of ubiquitous computing research at parc in the late 1980s," *IBM Syst. J.*, vol. 38, no. 4, pp. 693–696, dec 1999.

[5] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, ser. CVPR '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 886–893.

[6] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification (2Nd Edition)*. Wiley-Interscience, 2000.

[7] M. Kunneke, H. Stoltzing, C. Ohmann, Q. Yang, M. Künneke, H. Stöltzing, K. Thon, and W. Lorenz, "Bayes theorem and conditional dependence of symptoms: Different models applied to data of upper gastrointestinal bleeding," *Methods of Information in Medicine*, vol. 27, pp. 73–83, 1988.

[8] I. Kononenko, "Comparison of inductive and naive bayesian learning approaches to automatic knowledge acquisition," *Current trends in knowledge acquisition*, pp. 190–197, 1990.

[9] A. Gammerman and A. R. Thatcher, *Bayesian Inference in an Expert System without Assuming Independence*. New York, NY: Springer New York, 1990, pp. 182–218.

[10] B. S. Todd and R. Stamper, "A formal model of explanation," *Formal Aspects of Computing*, vol. 7, no. 2, pp. 207–225, 1995.

[11] B. Cestnik, I. Kononenko, and I. Bratko, "Assistant 86: A knowledge-elicitation tool for sophisticated users." in *EWSL*, 1987, pp. 31–45.

[12] S. Kung, *Digital Neural Networks*, ser. Information and system sciences series. Prentice Hall, 1993.

[13] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, ser. COLT '92. New York, NY, USA: ACM, 1992, pp. 144–152.

[14] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, Sep. 1995.

[15] M. Welling, "Fisher linear discriminant analysis," *Department of Computer Science, University of Toronto*, p. 4.

[16] K. Bailey, *Typologies and Taxonomies: An Introduction to Classification Techniques*, ser. Quantitative Applications in t. SAGE Publications, 1994, no. nÂ° 102.

[17] R. Tryon, *Cluster Analysis: Correlation Profile and Orthometric (factor) Analysis for the Isolation of Unities in Mind and Personality.* Edwards brother, Incorporated, lithoprinters and publishers, 1939.

[18] R. Cattell, *The Description of Personality: Basic Traits Resolved Into Clusters.* American psychological association, 1943.

[19] V. Estivill-Castro, "Why so many clustering algorithms: A position paper," *SIGKDD Explor. Newsl.*, vol. 4, no. 1, pp. 65–75, Jun. 2002.

[20] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise." AAAI Press, 1996, pp. 226–231.

[21] M. Ankerst, M. M. Breunig, H. peter Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure." ACM Press, 1999, pp. 49–60.

[22] O. Chapelle, B. Schlkopf, and A. Zien, *Semi-Supervised Learning*, 1st ed. The MIT Press, 2010.

[23] M. Baptista-Ríos, M. Marrón-Romera, C. Losada-Gutiérrez, J. A. Cruz-lozano, and A. del Abril, "Robust system for partially occluded people detection in RGB images," in *Proceedings of the 12th International Conference on Computer Vision Theory and Applications (VISAPP 2017). Aceptado para su publicación.*, 2017.

[24] J. Á. C. Lozano, "Detección y reconocimiento de personas en escenas naturales mediante visión artificial," Master, Alcala de Henares, 09/2014 2014.

[25] "Página oficial del programa KRename," http://www.krename.net/home/, Última visita: 22 de Septiembre de 2016.

[26] "Página oficial de OpenCV: documentación sobre HOG," http://docs.opencv.org/3.0-beta/modules/cuda/doc/object_detection.html, Última visita: 7 de Octubre de 2016.

[27] P. Joshi, D. Escriva, and V. Godoy, *OpenCV By Example.* Packt Publishing, 2016.

[28] "Página oficial de OpenCV: introducción a las SVM," http://docs.opencv.org/3.0-beta/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html, Última visita: 29 de Octubre de 2016.

[29] "Página oficial de OpenCV: documentación sobre SVM," http://docs.opencv.org/3.0-beta/modules/ml/doc/support_vector_machines.html, Última visita: 30 de Octubre de 2016.

[30] "Extracción y adecuación del vector soporte," http://software-against-developer.blogspot.com.es/2015/11/opencv-30-using-hogdescriptor-with-mlsvm.html, Última visita: 13 de Noviembre de 2016.

[31] "Ejemplo de la utilización de HOG y SVM en Matlab para la detección de caras en una imagen," http://thebrainiac1.blogspot.com.es/2012/07/v-behaviorurldefaultvmlo.html, Última visita: 5 de Diciembre de 2016.

Apéndice A

Manual de usuario

A.1 Introducción

Este apéndice ofrece un manual de usuario mediante el cual se puede hacer funcionar la aplicación desarrollada en el trabajo y reproducir los resultados obtenidos.

Esta aplicación ha sido programada tanto en lenguaje C++ como en *Matlab*. Para la desarrollada en C++, se hace uso de la librería de funciones *OpenCV* en su versión 3.0.0 y el compilador *gcc*, y para el sistema desarrollado en *Matlab* se utiliza el entorno *Matlab*R2016b.

Para la puesta en funcionamiento de ambos sistemas se debe usar el sistema operativo *Ubuntu 14.04.1 LTS* o superior.

A.2 Aplicación de C++

En la carpeta de la aplicación en C++ encontramos 5 subcarpetas que contienen los archivos necesarios para reproducir cada fase de la aplicación descrita en este proyecto. La carpeta *makefile* contiene los archivos necesarios para la compilación de cada una de las fases.

En dos de las carpetas, llamadas *HOG_descriptors* y *HOG_descriptors_neg*, se encuentran las imágenes de entrenamiento tanto positivas como negativas y los ejecutables que permiten extraer los descriptores de cada set a los archivos *Pos_descriptors.yml* y *Neg_descriptors.yml* respectivamente. Basta con ejecutar los programas *pos_imgs_descript* y *neg_imgs_descript* para obtener dichos archivos.

En otra de las carpetas, llamada *SVM_Train*, se encuentra el código que permite entrenar la SVM mediante los archivos .yml generados anteriormente y guardar el modelo para la siguiente fase, también en formato .yml. Basta con ejecutar el código *SVM_training* que se encuentra en la carpeta. Dada la cantidad de datos que maneja, cabe destacar que la ejecución de este programa requiere una cantidad de tiempo importante para completarse.

Por último, en la carpeta *Detection* está el código que implementa la fase de detección y una subcarpeta *Test* con imágenes utilizadas para probar el funcionamiento del sistema. Para reproducir los resultados obtenidos, basta con ejecutar el programa añadiendo como argumento la dirección de la imagen en la que queremos que se realice la detección, por ejemplo:

```
$ ./halfpeopledetector ./Test/imgprueba.png
```

Una vez ejecutado el programa, obtendremos un resultado como este:

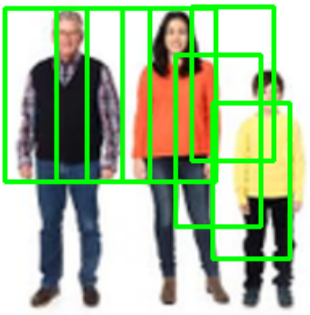

Figura A.1: Ejemplo de resultados obtenidos al ejecutar el programa *halfpeopledetector*.

Todas las fases de la aplicación han sido precompiladas, por lo que no hace falta compilarlas para poder ejecutarlas, aunque también se incluyen todos los archivos necesarios para su compilación.

A.3 Aplicación de Matlab

En la carpeta de la aplicación en *Matlab* encontramos dos subcarpetas y los archivos de extensión .m necesarios para ejecutar la aplicación.

La carpeta *Train* contiene a su vez dos carpetas en las que se encuentran las imágenes de entrenamiento positivas y negativas respectivamente. Aunque no son completamente necesarias para ejecutar el sistema de detección, ya que el archivo *SVM_64x128.mat* contiene un modelo de SVM ya entrenado con esas imágenes. La carpeta *Test* contiene un set de imágenes con las que probar el funcionamiento del sistema implementado.

Para reproducir los resultados obtenidos con en este proyecto basta con entrar en el programa *Matlab* y situarnos en la carpeta de la aplicación como carpeta de trabajo. Mediante el comando *Load* cargaremos la SVM ya entrenada. Para ejecutar la aplicación, basta con ejecutar la función *detect_in_image* con la imagen de la carpeta *Test* que nos interese analizar, y con el modelo de SVM previamente cargado. La figura A.2 muestra como se deben utilizar estos comandos.

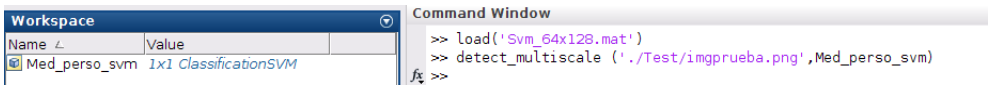

Figura A.2: Ejemplo de los comandos que se deben utilizar para ejecutar la aplicación.

Una vez el programa termine su ejecución, obtendremos un resultado como éste:

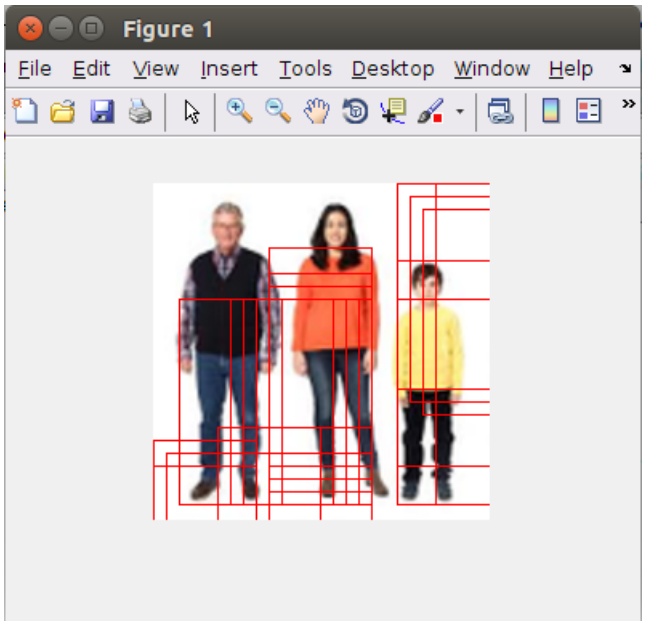
Figure 1
File Edit View Insert Tools Desktop Window Help

Apéndice B

Pliego de condiciones

Para la correcta utilización del sistema desarrollado en este trabajo, se debe disponer de un hardware y un software que cumpla ciertos requisitos mínimos.

B.1 Requisitos de Hardware

- Procesador de 32 ó 64 bits.
- 2GB de memoria RAM o superior.
- Al menos 1.5GB de memoria libres en el disco duro para funciones y datos.
- Al menos 1GB de memoria libre en el disco duro para la base de datos de INRIA.

B.2 Requisitos de Software

- Sistema operatvo *Linux Ubuntu 14.04.1 LTS*
- Librería *OpenCV 3.0.0*
- *Matlab R2016b.*
- Compilador *GNU GCC.*

Apéndice C

Presupuesto

C.1 Costes de equipamiento

- Equipamiento hardware utilizado:

Concepto	Cantidad	Coste Unitario	Subtotal(€)
PC HP G62	1	550€	550€
Coste total HW			550€

Tabla C.1: Coste del equipamiento hardware utilizado.

- Recursos software utilizados:

Concepto	Cantidad	Coste Unitario	Subtotal(€)
Ubuntu 14.04.1 LTS	1	0€	0€
Librería *OpenCV* 3.0.0	1	0€	0€
Software *Matlab* R2016b	1	4000€	4000€
Software LaTeX	1	0€	0€
Coste total SW			4000€

Tabla C.2: Coste de los recursos software utilizados.

C.2 Costes de mano de obra

Concepto	Cantidad (Horas)	Coste Unitario	Subtotal(€)
Desarrollo software	250	60€/hora	15.000€
Mecanografiado de la memoria	50	12€/hora	600€
Coste total de la mano de obra			15.600€

Tabla C.3: Coste de la mano de obra.

Concepto	Subtotal(€)
Equipamiento Hardware	550€
Recursos Software	4000€
Mano de obra	15.600€
Coste total del proyecto	20.150€

Tabla C.4: Coste total del proyecto.

C.3 Coste total del proyecto

El importe total del presupuesto asciende a la cantidad de: VEINTE MIL CIENTO CINCUENTA EUROS.

En Alcalá de Henares a _____ de _____ de 20_____

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá